

USENIX

conference

proceedings

Proceedings of the 2006 USENIX Annual Technical Conference

Boston, MA, USA, May 30–June 3, 2006

2006 USENIX Annual Technical Conference

Boston, MA, USA

May 30–June 3, 2006

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

The price is \$40 for members and \$50 for nonmembers.
Outside the U.S.A. and Canada, please add \$20 per copy for postage (via air printed matter).

Thanks to Our Sponsors



Thanks to Our Media Sponsors

ACM Queue
Addison-Wesley
Professional/Prentice
Hall Professional
Dr. Dobb's Journal
GRIDtoday

HPCwire
IEEE Security & Privacy
IDG World Expo
ITtoolbox
Linux Journal
No Starch Press

OSTG
SNIA
StorageNetworking.org
Sys Admin
UserFriendly.Org

© 2006 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-44-7

USENIX Association

**Proceedings of the
2006 USENIX Annual Technical Conference**

**May 30–June 3, 2006
Boston, MA, USA**

Conference Organizers

Program Chairs

Atul Adya, *Microsoft*

Erich Nahum, *IBM T.J. Watson Research Center*

Program Committee

Steven Bellovin, *Columbia University*

Ranjita Bhagwan, *IBM T.J. Watson Research Center*

Jeff Chase, *Duke University*

Mike Chen, *Intel Research, Seattle*

Jason Flinn, *University of Michigan*

Steven Hand, *University of Cambridge*

Gernot Heiser, *University of New South Wales and
National ICT Australia*

Kim Keeton, *Hewlett-Packard*

Dejan Kostić, *EPFL*

Jay Lepreau, *University of Utah*

Barbara Liskov, *Massachusetts Institute of Technology*

Jason Nieh, *Columbia University*

Vivek Pai, *Princeton University*

Dave Presotto, *Google*

John Reumann, *Google*

Mendel Rosenblum, *Stanford University*

Stefan Saroiu, *University of Toronto*

Geoff Voelker, *University of California, San Diego*

Alec Wolman, *Microsoft Research*

Yuanyuan Zhou, *University of Illinois at Urbana-
Champaign*

Invited Talks Committee

Chair: Christopher Small, *BBN Technologies*

Matt Blaze, *University of Pennsylvania*

The USENIX Association Staff

External Reviewers

Jeannie Albrecht	Archana Ganapathi	Jay Lorch	Reiner Sailer
Manish Anand	Dan Gebhardt	Luke Macpherson	Ken Salem
David G. Andersen	Craig Gentry	Anil Madhavapeddy	Geetanjali Sampemane
Eric Anderson	Chris Gniady	Kostas Magoutis	David Schultz
James W. Anderson	Ashvin Goel	Ratul Mahajan	Piyush Shivam
Henrique Andrade	Sachin Goyal	Varun Marupadi	Dan Simon
Boon S Ang	Charles Gray	Pankaj Mehra	Ian Sin
Ismail Ari	Ramakrishna Gummadi	Aravind Menon	Emin Gün Sirer
Godmar Back	Diwaker Gupta	Arif Merchant	Alex C. Snoeren
Ricardo Baratto	Andreas Haeberlen	Alan Mislove	David Snowden
Stefan Berger	Tim Harris	Vishal Misra	Tim Sohn
Ricardo Bianchini	Alejandro Hevia	Iqbal Mohamed	Jonathan Stone
Bill Bolosky	Mike Hibler	Justin Moore	Adam Stubblefield
Tanya Bragin	Kirsten Hildrum	Alexander Moshchuk	Jing Su
Ryan Braud	Anne M. Holler	Daniel Myers	Ya-Yunn Su
Anton Burtsev	Jon Howell	Moni Naor	Pratap Subrahmanyam
Simon Byers	Bradley Huffaker	Vivek Narasayya	Lakshminarayanan
Ranveer Chandra	Galen Hunt	Edmund B. Nightingale	Subramanian
Matthew Chapman	John Ioannidis	David Olshefski	Jeremy Sugerman
Peter Chen	Anca Ivan	David Oppenheimer	Ram Swaminathan
Yuqun Chen	Hani Jamjoom	Scott Owens	Renata Teixeira
Winnie Cheng	Song Jiang	Jitu Padhye	Marvin Theimer
Lucy Cherkasova	Charanjit Jutla	Venkat Padmanabhan	Andy Tucker
Peter Chubb	Magnus Karlsson	KyoungSoo Park	Volkmar Uhlig
Rebecca Collins	Vishal Kathuria	Yoonho Park	Mustafa Uysal
Nathan Coopridge	Angelos Keromytis	Daniel Peek	Ben Vandiver
James Cowling	Chip Killian	Trevor Pering	Alex Varshavsky
Olivier Crameri	Nick Koudas	Ben Pfaff	Rob von Behren
Anwitaman Datta	Dmitri Krioukov	Dan Phung	Carl Waldspurger
Colin Dixon	Joanna Kulik	Mike Piatek	Helen Wang
John R. Douceur	Ihor Kuz	Jesse Pool	Limin Wang
Fred Douglass	Oren Laadan	Ansley Post	Andrew Warfield
Steven Dropsho	Jeremy Lau	Shaya Potter	Kirk Webb
John Dunagan	Alvin R. Lebeck	Prashanth Radhakrishnan	Simon Winwood
Mathilde Durvy	Ben Leong	Venugopalan	Charles P. Wright
Eric Eide	Joshua LeVasseur	Ramasubramanian	Jay J. Wylie
Kevin Elphinstone	Philip Levis	Kavitha Ranganathan	Kenneth G. Yocum
Nick Feamster	Baochun Li	Felix Rauch	Erez Zadok
Nicholas FitzRoy-Dale	David Lie	John Regehr	Qingbo Zhu
Matthew Flatt	Lionel Litty	Charles Reis	Cliff Zou
Keir Fraser	Jiuxing Liu	Patrick Reynolds	
Jon Froehlich	Michael Locasto	David Richardson	

2006 USENIX Annual Technical Conference
May 30–June 3, 2006
Boston, MA, USA

Index of Authors	vii
Message from the Program Chairs	ix

Thursday, June 1, 2006

Virtualization

Antfarm: Tracking Processes in a Virtual Machine Environment.	1
<i>Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison</i>	
Optimizing Network Virtualization in Xen	15
<i>Aravind Menon, EPFL; Alan L. Cox, Rice University; Willy Zwaenepoel, EPFL</i>	
High Performance VMM-Bypass I/O in Virtual Machines	29
<i>Jiuxing Liu, IBM T.J. Watson Research Center; Wei Huang, The Ohio State University; Bulent Abali, IBM T.J. Watson Research Center; Dhabaleswar K. Panda, The Ohio State University</i>	

Storage

Provenance-Aware Storage Systems	43
<i>Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer, Harvard University</i>	
Thresher: An Efficient Storage Manager for Copy-on-write Snapshots	57
<i>Liuba Shrira and Hao Xu, Brandeis University</i>	
Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines	71
<i>Partho Nath, Penn State University; Michael A. Kozuch, Intel Research Pittsburgh; David R. O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia, and Matt Touns, Carnegie Mellon University</i>	

Short Papers Session I

Compare-by-Hash: A Reasoned Analysis	85
<i>J. Black, University of Colorado, Boulder</i>	
An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems	91
<i>Paul Willmann, Scott Rixner, and Alan L. Cox, Rice University</i>	
Disk Drive Level Workload Characterization	97
<i>Alma Riska and Erik Riedel, Seagate Research</i>	
Towards a Resilient Operating System for Wireless Sensor Networks	103
<i>Hyoseung Kim and Hojung Cha, Yonsei University</i>	
Transparent Contribution of Memory	109
<i>James Cipar, Mark D. Corner, and Emery D. Berger, University of Massachusetts, Amherst</i>	

Friday, June 2, 2006

Server Implementation

- Implementation and Evaluation of Moderate Parallelism in the BIND9 DNS Server 115
Tatuya Jinmei, Toshiba Corporation; Paul Vixie, Internet Systems Consortium
- Flux: A Language for Programming High-Performance Servers 129
Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D. Berger, and Mark D. Corner, University of Massachusetts, Amherst
- Understanding and Addressing Blocking-Induced Network Server Latency 143
Yaoping Ruan, IBM T.J. Watson Research Center; Vivek Pai, Princeton University

Security

- Reval: A Tool for Real-time Evaluation of DDoS Mitigation Strategies 157
Rangarajan Vasudevan and Z. Morley Mao, University of Michigan; Oliver Spatscheck and Jacobus van der Merwe, AT&T Labs—Research
- LADS: Large-scale Automated DDoS Detection System 171
Vyas Sekar, Carnegie Mellon University; Nick Duffield, Oliver Spatscheck, and Jacobus van der Merwe, AT&T Labs—Research; Hui Zhang, Carnegie Mellon University
- Bump in the Ether: A Framework for Securing Sensitive User Input 185
Jonathan M. McCune, Adrian Perrig, and Michael K. Reiter, Carnegie Mellon University

Management and Administration

- Sharing Networked Resources with Brokered Leases 199
David Irwin, Jeff Chase, Laura Grit, Aydan Yumerefendi, and David Becker, Duke University; Kenneth G. Yocum, University of California, San Diego
- Understanding and Validating Database System Administration 213
Fábio Oliveira, Kiran Nagaraja, Rekha Bachwani, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen, Rutgers University
- SMART: An Integrated Multi-Action Advisor for Storage Systems 229
Li Yin, University of California, Berkeley; Sandeep Uttamchandani, Madhukar Korupolu, and Kaladhar Voruganti, IBM Almaden Research Center; Randy Katz, University of California, Berkeley

Short Papers Session II

- sMonitor: A Non-Intrusive Client-Perceived End-to-End Performance Monitor of Secured Internet Services .. 243
Jianbin Wei and Cheng-Zhong Xu, Wayne State University
- Privacy Analysis for Data Sharing in *nix Systems 249
Aameek Singh, Ling Liu, and Mustaque Ahamad, Georgia Institute of Technology
- Securing Web Service by Automatic Robot Detection 255
KyoungSoo Park and Vivek S. Pai, Princeton University; Kang-Won Lee and Seraphin Calo, IBM T.J. Watson Research Center
- Cutting through the Confusion: A Measurement Study of Homograph Attacks 261
Tobias Holgers, David E. Watson, and Steven D. Gribble, University of Washington
- Stealth Probing: Efficient Data-Plane Security for IP Routing 267
Ioannis Avramopoulos and Jennifer Rexford, Princeton University

Saturday, June 3, 2006

Wide Area Distributed Systems

- Service Placement in a Shared Wide-Area Platform 273
David Oppenheimer, University of California, San Diego; Brent Chun, Arched Rock Corporation; David Patterson, University of California, Berkeley; Alex C. Snoeren and Amin Vahdat, University of California, San Diego
- Replay Debugging for Distributed Applications 289
Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica, University of California, Berkeley
- Loose Synchronization for Large-Scale Networked Systems 301
Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat, University of California, San Diego

Network and Operating System Support

- System- and Application-level Support for Runtime Hardware Reconfiguration on SoC Platforms 315
Dimitris Syrivellis and Spyros Lalis, University of Thessaly, Hellas
- Resilient Connections for SSH and TLS 329
Teemu Koponen, Helsinki Institute for Information Technology; Pasi Eronen, Nokia Research Center; Mikko Särelä, Helsinki University of Technology
- Structured and Unstructured Overlays under the Microscope: A Measurement-based View of Two P2P Systems That People Use 341
Yi Qiao and Fabián E. Bustamante, Northwestern University

Short Papers Session III

- Reclaiming Network-wide Visibility Using Ubiquitous Endsystem Monitors 357
Evan Cooke, University of Michigan; Richard Mortier, Austin Donnelly, Paul Barham, and Rebecca Isaacs, Microsoft Research, Cambridge
- Integrated Scientific Workflow Management for the Emulab Network Testbed 363
Eric Eide, Leigh Stoller, Tim Stack, Juliana Freire, and Jay Lepreau, University of Utah
- How DNS Misnaming Distorts Internet Topology Mapping 369
Ming Zhang, Microsoft Research; Yaoping Ruan, IBM Research; Vivek Pai and Jennifer Rexford, Princeton University
- Efficient Query Subscription Processing for Prospective Search Engines 375
Utku Irmak, Polytechnic University; Svilen Mihaylov, University of Pennsylvania; Torsten Suel, Polytechnic University; Samrat Ganguly and Rauf Izmailov, NEC Laboratories America
- IP Only Server 381
Muli Ben-Yehuda, Oleg Goldshmidt, Elliot K. Kolodner, Zorik Machulsky, Vadim Makhervaks, Julian Satran, Marc Segal, Leah Shalev, and Ilan Shimony, IBM Haifa Research Laboratory

Index of Authors

Abali, Bulent	29	Isaacs, Rebecca	357	Rixner, Scott	91
Ahamad, Mustaque	249	Izmailov, Rauf	375	Ruan, Yaoping	143, 369
Albrecht, Jeannie	301	Jinmei, Tatuya	115	Särelä, Mikko	329
Altekar, Gautam	289	Jones, Stephen T.	1	Satran, Julian	381
Arpaci-Dusseau, Andrea C.	1	Katz, Randy	229	Satyanarayanan, M.	71
Arpaci-Dusseau, Remzi H.	1	Kim, Hyoseung	103	Segal, Marc	381
Avramopoulos, Ioannis	267	Kolodner, Elliot K.	381	Sekar, Vyas	171
Bachwani, Rekha	213	Koponen, Teemu	329	Seltzer, Margo	43
Barham, Paul	357	Korupolu, Madhukar	229	Shalev, Leah	381
Becker, David	199	Kostadinov, Alexander	129	Shenker, Scott	289
Ben-Yehuda, Muli	381	Kozuch, Michael A.	71	Shimony, Ilan	381
Berger, Emery D.	109, 129	Lalis, Spyros	315	Shrira, Liuba	57
Bianchini, Ricardo	213	Lee, Kang-Won	255	Singh, Aameek	249
Black, J.	85	Lepreau, Jay	363	Snoeren, Alex C.	273, 301
Braun, Uri	43	Liu, Jiuxing	29	Spatscheck, Oliver	157, 171
Burns, Brendan	129	Liu, Ling	249	Stack, Tim	363
Bustamante, Fabián E.	341	Machulsky, Zorik	381	Stoica, Ion	289
Calo, Seraphin	255	Makhervaks, Vadim	381	Stoller, Leigh	363
Cha, Hojung	103	Mao, Z. Morley	157	Suel, Torsten	375
Chase, Jeffrey	199	Martin, Richard P.	213	Syrivelis, Dimitris	315
Chun, Brent	273	McCune, Jonathan M.	185	Tolia, Niraj	71
Cipar, James	109	Menon, Aravind	15	Toups, Matt	71
Cooke, Evan	357	Mihaylov, Svilen	375	Tuttle, Christopher	301
Corner, Mark D.	109, 129	Mortier, Richard	357	Uttamchandani, Sandeep	229
Cox, Alan L.	15, 91	Muniswamy-Reddy, Kiran-Kumar	43	Vahdat, Amin	273, 301
Donnelly, Austin	357	Nagaraja, Kiran	213	van der Merwe, Jacobus	157, 171
Duffield, Nick	171	Nath, Partho	71	Vasudevan, Rangarajan	157
Eide, Eric	363	Nguyen, Thu D.	213	Vixie, Paul	115
Eronen, Pasi	329	O'Hallaron, David R.	71	Voruganti, Kaladhar	229
Freire, Juliana	363	Oliveira, Fábio	213	Watson, David E.	261
Ganguly, Samrat	375	Oppenheimer, David	273	Wei, Jianbin	243
Geels, Dennis	289	Pai, Vivek S.	143, 255, 369	Willmann, Paul	91
Goldshmidt, Oleg	381	Panda, Dhabaleswar K.	29	Xu, Cheng-Zhong	243
Gribble, Steven D.	261	Park, KyoungSoo	255	Xu, Hao	57
Grimaldi, Kevin	129	Patterson, David	273	Yin, Li	229
Grit, Laura	199	Perrig, Adrian	185	Yocum, Kenneth G.	199
Harkes, Jan	71	Qiao, Yi	341	Yumerefendi, Aydan	199
Holgers, Tobias	261	Reiter, Michael K.	185	Zhang, Hui	171
Holland, David A.	43	Rexford, Jennifer	267, 369	Zhang, Ming	369
Huang, Wei	29	Riedel, Erik	97	Zwaenepoel, Willy	15
Irmak, Utku	375	Riska, Alma	97		
Irwin, David	199				

Message from the Program Chairs

It's our great pleasure to present the program for the 2006 USENIX Annual Technical Conference. The USENIX ATC continues its decades-long history of providing a venue for fine experimental systems work, spanning a wide range of areas such as operating systems, networking, storage, security, distributed systems, virtualization, and more. This year continues the short papers track that was introduced last year and found to be very successful.

The program you see here is the result of a record number of submissions: 153 full papers and 41 short papers. To handle these, we generated 579 reviews averaging 48 lines of text, with the longest review measuring 426 lines. In the end, 21 papers were accepted as full papers and 15 were accepted as short papers. Thus, many good papers could not be accepted.

Many, many people contributed to the conference, and we would like to express our thanks and appreciation to them. First is to the technical program committee. On top of the record number of submissions, we had an accelerated review schedule. Each PC member (including the program chairs) had an average of 26 papers to review in about 3 weeks. We are thus especially grateful to the PC for doing such an excellent job under such hurried conditions. We expect that next year's USENIX PC will have a more normal reviewing schedule.

We would also like to thank the many external reviewers for their help, also listed in these proceedings, and the great support we had from the USENIX staff, including Casey Henderson, Jennifer Joost, Jane-Ellen Long, and Ellie Young.

Finally, we would like to thank the authors of all the submitted papers. Without you, there would be no conference.

We hope you enjoy the program!

Atul Adya and Erich Nahum
Co-Chairs, 2006 USENIX Annual Technical Conference

Antfarm: Tracking Processes in a Virtual Machine Environment

Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Department of Computer Sciences
University of Wisconsin, Madison
{stjones,dusseau,remzi}@cs.wisc.edu

Abstract

In a virtualized environment, the VMM is the system's primary resource manager. Some services usually implemented at the OS layer, like I/O scheduling or certain kinds of security monitoring, are therefore more naturally implemented inside the VMM. Implementing such services at the VMM layer can be complicated by the lack of OS and application-level knowledge within a VMM. This paper describes techniques that can be used by a VMM to independently overcome part of the "semantic gap" separating it from the guest operating systems it supports. These techniques enable the VMM to track the existence and activities of *operating system processes*. Antfarm is an implementation of these techniques that works without detailed knowledge of a guest's internal architecture or implementation. An evaluation of Antfarm for two virtualization environments and two operating systems shows that it can accurately infer process events while incurring only a small 2.5% runtime overhead in the worst case. To demonstrate the practical benefits of process information in a VMM we implement an anticipatory disk scheduler at the VMM level. This case study shows that significant disk throughput improvements are possible in a virtualized environment by exploiting process information within a VMM.

1 Introduction

Virtual machine technology is increasingly being deployed on a range of platforms from high-end servers [4, 24, 25] to desktop PCs [22]. There is a large and growing list of reasons to use virtualization in these diverse computing environments, including server consolidation [25], support for multiple operating systems (including legacy systems) [11], sandboxing and other security features [9, 16], fault tolerance [3], and optimization for specialized architectures [4]. As both software [6] and hardware support [12, 13] for zero-overhead virtualization develops, and as virtualization is included in dominant

commercial operating systems [2], we expect virtualized computing environments to become nearly ubiquitous.

As virtualization becomes prevalent, the *virtual machine monitor* (VMM) naturally supplants the operating system as the primary resource manager for a machine. Where one used to consider the OS the main target for innovation in system services, one should now consider how to implement some of those services within a VMM [5].

The transition of some functionality from the OS into the VMM has many potential benefits. For example, by implementing a feature a single time within a VMM, it becomes available to *all* operating systems running above. Further, the VMM may be the only place where new features can be introduced into a system, as the operating system above is legacy or closed-source or both. Finally, the VMM is the *only* locale in the system that has total control over system resources and hence can likely make the most informed resource management decisions.

However, pushing functionality down one layer in the software stack into the VMM has its drawbacks as well. One significant problem is the lack of higher-level knowledge within the VMM, sometimes referred to as a *semantic gap* [5]. Previous work in virtualized environments has partially recognized this dilemma, and researchers have thus developed techniques to infer higher-level *hardware resource utilization* [4, 20, 24]. These techniques are useful because they allow a VMM to better manage the resources of the system, (*e.g.*, by reallocating an otherwise idle page in one virtual machine to a different virtual machine that could use it [24]).

In addition, some recently proposed VMM-based services use explicit information about the *software abstractions* of the operating systems running above them to bridge the semantic gap [10, 15]. However, previous work has not thoroughly explored how a VMM can *learn* about the software abstractions of the operating systems running above without the information being given explicitly to it. Being able to implicitly learn about operating systems from within a VMM is important if a guest OS is proprietary, untrusted, or is managed by a different entity than

the one managing the VMM. In these cases, explicit information about the details of the guest's memory layout or implementation will be unavailable or unreliable.

In this paper, we develop a set of techniques that enable a virtual machine monitor to implicitly discover and exploit information about one of the most important operating system abstractions, the *process*. By monitoring low-level interactions between guest operating systems and the memory management structures on which they depend, we show that a VMM can accurately determine when a guest operating system creates processes, destroys them, or context-switches between them. These techniques operate without any explicit information about the guest operating system vendor, version, or implementation details.

We demonstrate the utility and efficacy of VMM-level process awareness by building an anticipatory disk scheduler [14] within a VMM. In a virtual machine environment, an anticipatory disk scheduler requires information from both the VMM and OS layers and so cannot be implemented exclusively in either. Making a VMM process aware overcomes this limitation and allows an OS-neutral implementation at the VMM layer without any modifications or detailed knowledge of the OS above. Our implementation within the VMM is able to improve throughput among competing sequential streams from processes across different virtual machines or within a single guest operating system by a factor of two or more.

In addition to I/O scheduling, process information within the VMM has several other immediate applications, especially in the security domain. For example, it can be used to detect that processes have been hidden from system monitoring tools by malicious software or to identify code and data from sensitive processes that should be monitored for runtime modification [10]. Patterns of system calls associated with a process can be used to recognize when a process has been compromised [8, 19]. In addition to detection, techniques exist to slow or thwart intrusions at the process level by affecting process scheduling [21]. Finally, process information can be used as the basis for discovering other high-level OS abstractions. For example, the parent-child relationship between processes can be used to identify groups of related processes associated with a *user*. All of these applications are feasible within a VMM only when process information is available.

Antfarm is the implementation of our process identification techniques for two different virtualization environments, Xen and Simics. Antfarm has been evaluated as applied to x86/Linux, x86/Windows, and SPARC/Linux guest operating systems. This range of environments spans two processor families with significantly different virtual memory management interfaces and two operating systems with very different process management semantics. Antfarm imposes only a small runtime overhead of

about 2.4% in a worst case scenario and about 0.6% in a more common, process-intensive compilation environment.

The rest of the paper is organized as follows. In Section 2 we place Antfarm in context with related work. Then in Section 3, we cover some required background material relating to our implementation architectures and virtual machines in general. This is followed in Section 4 by a discussion of the techniques underlying Antfarm. Section 5 covers the implementation details of Antfarm. We evaluate the accuracy and overhead imposed by Antfarm in Section 6. In Section 7, we present our anticipatory scheduling case study and then conclude in Section 8.

2 Related Work

Antfarm informs a VMM about one important operating system abstraction, the process, about which it would otherwise have no information. Other research has recognized that information not explicitly available to a VMM is nevertheless useful when implementing VMM features and services.

In some cases the information relates to hardware. Disco [4], for example, determines when the guest is executing in its idle loop by detecting when it enters a low-power processor mode. VMWare's ESX Server [24] uses page sampling to determine the utilization of physical memory assigned to each of its virtual machines. Antfarm differs from these efforts in that it focuses on inferring information about processes, a software construct.

Other projects have also recognized the value of OS-level information in a VMM. In some cases, detailed version-specific memory layout information as well as the semantic knowledge to make use of that information has been exported directly to the VMM. VMI [10] does this to implement security techniques like detecting malicious, hidden processes within a guest. IntroVirt [15] uses memory layout and implementation details to enable novel host-based intrusion detection features in a VMM. Antfarm, in contrast, enables a more limited and inexact level of information to be inferred by a VMM. It does this, however, without any explicit information about memory layout or implementation of affected guests and so can be deployed in a broader set of environments.

Work by Uhlig *et al.* [23] is more similar to our own. It shows how to infer guest-level information to do processor management more intelligently in a multiprocessor environment. Specifically, they deduce when no kernel locks are held by observing when the OS above is executing in user versus kernel mode. Antfarm is complementary. It observes a different virtual resource, the MMU, to infer information about operating system processes.

Finally, as an alternative to inferring OS-level information, such knowledge could be passed explicitly from the OS to the VMM, as is done, (to some extent), in paravirtualized architectures [6, 25]. Explicit information supplied by a paravirtualized OS is guaranteed to match what is available inside the OS. By this metric, paravirtual information should be considered the gold standard of OS information within a VMM. In some important environments, however, the explicit approach is less valuable. For example, paravirtualization requires OS-level modification, which implies that functionality cannot be deployed in VMM's running beneath legacy or closed-source operating systems. For the same reasons, dependence on explicit interfaces forces innovation in the VMM that requires OS-level information to be coupled with changes to supported operating systems. Inferring guest information allows a VMM to innovate independent of the OS implementation. Finally, in the case of security applications, a guest OS cannot be trusted to report on its own activities using a paravirtualized interface because it may have been compromised and intentionally mislead the VMM.

3 Background

The techniques we describe in this paper are based on the observations that a VMM can make of the interactions between a guest OS and virtual hardware. Specifically, Antfarm monitors how a guest uses a virtual MMU to implement virtual address spaces. In this section we review some of the pertinent details of the Intel x86 and the SPARC architectures used by Antfarm. We also discuss some basic features of virtual machine monitors and the runtime information available to them.

3.1 x86 Virtual Memory Architecture

Our first implementation platform is the Intel x86 family of microprocessors. We chose the x86 because it is the most frequently virtualized processor architecture in use today. This section reviews the features of the x86 virtual memory architecture that are important for our inference techniques.

The x86 architecture uses a two-level, in-memory, architecturally-defined page table. The page table is organized as a tree with a single 4 KB memory page called the *page directory* at its root. Each 4-byte entry in the page directory can point to a 4 KB page of the *page table* for a process.

Each page table entry (PTE) that is in active use contains the address of a physical page for which a virtual mapping exists. Various page protection and status bits are also available in each PTE that indicate, for example, whether a page is writable or whether access to a page is

restricted to privileged software.

A single address space is active per processor at any given time. System software informs the processor's MMU that a new address space should become active by writing the physical address of the page directory for the new address space into a processor control register (CR3). Since access to this register is privileged the VMM must virtualize it on behalf of guest operating systems.

TLB entries are loaded on-demand from the currently active page tables by the processor itself. The operating system does not participate in handling TLB misses.

An operating system can explicitly remove entries from a TLB in one of two ways. A single entry can be removed with the `INVLPG` instruction. All non-persistent entries (those entries whose corresponding page table entries are not marked "global") can be flushed from the TLB by writing a new value to CR3. Since no address space or process ID tag is maintained in the TLB, all non-shared entries must be flushed on context switch.

3.2 SPARC Virtual Memory Architecture

In this section we review the key aspects of the SPARC MMU, especially how it differs from the x86. We chose the SPARC as our second implementation architecture because it provides a significantly different memory management interface to system software than the x86.

Instead of architecturally-defined, hardware-walked page tables as on the x86, SPARC uses a software managed TLB, *i.e.*, system software implements virtual address spaces by explicitly managing the contents of the hardware TLB. When a memory reference is made for which no TLB entry contains a translation, the processor raises an exception, which gives the operating system the opportunity to supply a valid translation or deliver an error to the offending process. The CPU is not aware of the operating system's page table organization.

In order to avoid flushing the entire TLB on process context switches, SPARC supplies a tag for each TLB entry, called a *context ID*, that associates the entry with a specific virtual address space. For each memory reference, the current context is supplied to the MMU along with the desired virtual address. In order to match, both the virtual page number and context in a TLB entry must be identical to the supplied values. This allows entries from distinct address spaces to exist in the TLB simultaneously.

An operating system can explicitly remove entries from the TLB at the granularity of a single page or at the granularity of an entire address space. These operations are called `page demap` and `context demap` respectively.

3.3 Virtual Machines

A VMM implements a hardware interface in software. The interface includes the privileged, or system, portions of the microprocessor architecture as well as peripherals like disk, network, and user interface devices. Note that the non-privileged, or user, portion of the microprocessor instruction set is not virtualized; when running unprivileged instructions, the guest directly executes on the processor with no additional overhead.

A key feature of a virtualized system environment is that guest operating systems execute using the unprivileged mode of the processor, while the VMM runs with full privilege. All guest OS accesses to sensitive system components, like the MMU or I/O peripherals, cause the processor to trap to the VMM. This allows the VMM to virtualize sensitive system features by mediating access to the feature or emulating it entirely. For example, because the MMU is virtualized, all attempts by a guest operating system to establish a virtual-to-physical memory mapping are trapped by the VMM; hence, the VMM can observe all such attempts. Similarly, each request to a virtual disk device is available for a VMM to examine. The VMM can choose to service a request made via a virtualized interface in any way it sees fit. For example, requests for virtual mappings can be altered or disk requests can be reordered.

4 Process Identification

The key to our process inference techniques is the logical correspondence between the abstraction *process*, which is not directly visible to a VMM, and the *virtual address space*, which is. This correspondence is due to the traditional single address space per process paradigm shared by all modern operating systems.

There are three major process events we seek to observe: creation, exit, and context switch. To the extent address spaces correspond to processes, these events are approximated by address space creation, destruction, and context switch. Hence, our techniques track processes by tracking address spaces.

Our approach to tracking address spaces on both x86 and SPARC is to identify a VMM-visible value with which we can associate a specific address space. We call this value an address space identifier (ASID). Tracking address space creation and context switch then becomes simply observing the use of a particular piece of VMM-visible operating system state, the ASID.

For example, when an ASID is observed that has not been seen before, we can infer that a new address space has been created. When one ASID is replaced by another ASID, we can conclude that an address space context switch has occurred. The technique we use to identify

address space deallocation consists of detecting when an ASID is available for reuse. We assume that the address space, to which an ASID refers, has been deallocated if its associated ASID is available for reuse.

4.1 Techniques for x86

On the x86 architecture we use the physical address of the page directory as the ASID. A page directory serves as the root of the page table tree that describes each address space. The address of the page directory is therefore characteristic of a single address space.

4.1.1 Process Creation and Context Switch

To detect address space creation on x86 we observe how page directories are used. A page directory is in use when its physical address resides in CR3. The VMM is notified whenever a guest writes a new value to CR3 because it is a privileged register. If we observe an ASID value being used that has not been seen before, we can infer that a new address space has been created. When an ASID is seen for the first time, the VMM adds it to an ASID registry, akin to an operating system process list, for tracking purposes.

Writes to CR3 also imply address space context switch. By monitoring these events, the VMM always knows which ASID is currently “active”.

4.1.2 Process Exit

To detect address space deallocation, we use knowledge about the generic responsibilities of an operating system to maintain address space isolation. These requirements lead to distinctive OS behavior that can be observed and exploited by a VMM to infer when an address space has been destroyed.

Operating systems must strictly control the contents of page tables being used to implement virtual address spaces. Process isolation could be breached if a page directory or page table page were reused for distinct processes without first being cleared of their previous entries. To ensure this invariant holds, Windows and Linux systematically clear the non-privileged portions of page table pages used by a process prior to reusing them. Privileged portions of the page tables used to implement the protected kernel address space need not be cleared because they are shared between processes and map memory not accessible to untrusted software.

An OS must also ensure that no stale entries remain in any TLB once an address space has been deallocated. Since the x86 architecture does not provide a way for entries from multiple address spaces to coexist in a TLB, a TLB must be completely flushed prior to reusing address space structures like the page directory. On x86, the TLB

is flushed by writing a value to CR3, an event the VMM can observe.

Hence, to detect user address space deallocation, a VMM can keep a count of the number of user virtual mappings present in the page tables describing an address space. When this count drops to zero, the VMM can infer that one requirement for address space reuse has been met. It is simple for a VMM to maintain such a counter because the VMM *must* be informed of all updates to a process's page tables in order for those updates to be effective. This requirement follows from the VMM's role in virtualizing the MMU. Multi-threading does not introduce additional complexity, because updates to a process's page tables must always be synchronized within the VMM for correctness.

By monitoring TLB flushes on all processors, a VMM can detect when the second requirement for address space deallocation has been met. Once both events have been observed for a particular ASID, the VMM can consider the corresponding address space dead and its entry in the ASID registry can be removed. A subsequent use of the same ASID implies the creation of a new and distinct process address space.

4.2 Techniques for SPARC

The key aspect that was used to enable process awareness on x86 is still present on SPARC. Namely, there is a VMM-visible identifier associated with each virtual address space. On x86 this was the physical address of the page directory. On SPARC we use the virtual address space context ID as an ASID. Making the obvious substitution leads to a process detection technique for SPARC similar to that for x86.

4.2.1 Creation and Context Switch

On SPARC, installing a new context ID is a privileged operation and so it is always visible to a VMM. By observing this operation, a VMM can maintain a registry of known ASIDs. When a new ASID is observed that is not in the ASID registry, the VMM infers the creation of a new address space. Context switch is detected on SPARC when a new context ID is installed on a processor.

4.2.2 Exit

The only requirement for the reuse of a context ID on SPARC is that all stale entries from the previously associated address space be removed from each processor's TLBs. SPARC provides the context demap operation for this purpose. Instead of monitoring page table contents, as on x86, a VMM can observe context demap operations. If all entries for a context ID have been flushed from every

	x86	SPARC
ASID	Page directory PA	Context ID
Creation	New ASID	New ASID
Exit	No user mappings and TLB flushed	Context demap
Context switch	CR3 change	Context ID change

Table 1: **Process identification techniques.** The table lists the techniques used by Antfarm to detect each process event on the x86 and SPARC architectures.

processor it implies that the associated address space is no longer valid.

5 Implementation

Antfarm has been implemented for two virtualization environments. The first, Xen [6], is a true VMM. The other is a low-level system simulator called Simics [17] which we use to explore process awareness for operating systems and architectures not supported by Xen.

5.1 Antfarm for Xen

Xen is an open source virtual machine monitor for the Intel x86 architecture. Xen provides a paravirtualized [25] processor interface, which enables lower overhead virtualization at the expense of porting system software. We explicitly do *not* make use of this feature of Xen; hence, the mechanisms we describe are equally applicable to a more conventional virtual machine monitor such as VMWare [22, 24]. Because operating systems must be ported to run on Xen, proprietary commercial operating systems like Microsoft Windows are not currently supported.

Antfarm for Xen is implemented as a set of patches to the Xen hypervisor. Changes are concentrated in the handlers for events like page faults, page table updates, and privileged register access. Additional hooks were added to Xen's back-end block device driver. The Antfarm patches to Xen, including debugging and measurement infrastructure, total approximately 1200 lines across eight files.

5.2 Antfarm for Simics

Simics [17] is a full system simulator capable of executing unmodified, commercial operating systems and applications for a variety of processor architectures. While Simics is not a virtual machine monitor in the strict sense of direct execution of user instructions [18], it can play the

role of a VMM by allowing Antfarm to observe and interpose on operating system and application hardware requests in the same way a VMM does. Simics allows us to explore process awareness techniques for SPARC/Linux and x86/Windows which would not be possible with a Xen-only implementation.

Antfarm for Simics is implemented as a Simics extension module. Simics extension modules are shared libraries dynamically linked with the main Simics executable. Extension modules can read or write OS and application memory and registers in the same way as a VMM.

Simics provides hooks called “hops” for various hardware events for which extension modules can register callback functions. Antfarm for Simics/x86 uses a hop to detect writes to CR3 and Antfarm for Simics/SPARC uses a hop to detect when the processor context ID is changed. Invocation of a callback is akin to the exception raised when a guest OS accesses privileged processor registers on a true VMM. A memory write breakpoint is installed by Antfarm for Simics/x86 on all pages used as page tables so that page table updates can be detected. A VMM like Xen marks page tables read-only to detect the same event.

Antfarm for Simics/x86 consists of about 800 lines of C code. For Simics/SPARC the total is approximately 450 lines.

6 Process Awareness Evaluation

In this section we explore the accuracy of Antfarm in each of our implementation environments. We also characterize the runtime overhead of Antfarm for Xen.

The analysis of accuracy can be decomposed into two components. The first is the ability to correctly detect process creations, exits, and context switches. We call this aspect *completeness*. The second component is the time difference or *lag* between process events as they occur within the operating system and when they are detected by the VMM.

6.1 x86 Evaluation

Our evaluation on x86 uses Xen version 2.0.6. Version 2.6.11 of the Linux kernel was used in Xen’s privileged control VM. Linux kernel version 2.4.30 and 2.6.11 are used in unprivileged VMs as noted. Our evaluation hardware consists of a 2.4 GHz Pentium IV PC with 512 MB of RAM. Virtual machines are each allocated 128 MB of RAM in this environment.

We also evaluate our techniques as applied to Microsoft Windows NT4. Since Windows does not currently run on Xen, Simics/x86 is used for this purpose. Our Simics/x86

virtual machines were configured with a 2.4 GHz Pentium IV and 256 MB of RAM.

6.1.1 Completeness

To quantify completeness, each guest operating system was instrumented to report process creation, exit, and context switch. Event records include the appropriate ASID, as well as the time of the event obtained from the processor’s cycle counter. These OS traces were compared to similar traces generated by Antfarm. Guest OS traces are functionally equivalent to the information that would be provided by a paravirtualized OS that included a process event interface. Hence, our evaluation implicitly compares the accuracy of Antfarm to the ideal represented by a paravirtual interface.

In addition to process creation, exit, and context switch, guests report address space creation and destruction events so that we can discriminate between errors caused by a mismatch between processes and address spaces and errors induced by inaccurate address space inferences made by Antfarm.

We categorize incorrect inferences as either false negatives or false positives. A false negative occurs when a true process event is missed by Antfarm. A false positive occurs when Antfarm incorrectly infers events that do not exist.

To determine if false negatives occurred, one-to-one matches were found for every OS-reported event in each pair of traces. We required that the matching event have the same ASID, and that it occur within the range for which the event was plausible. For example, to match an OS process-creation event, the corresponding inferred event must occur after any previous OS-reported process exit events with the same ASID and before any subsequent OS-reported process creation events with the same ASID.

Table 2 reports the process and address space event counts gathered by our guest OSes and by Antfarm during an experiment utilizing two process intensive workloads. The first workload is synthetic. It creates 1000 processes, each of which runs for 10 seconds then exits. The process creation rate is 10 processes/second. On Linux, this synthetic workload has three variants. The first creates processes using fork only; the second uses fork followed by exec; the third employs vfork followed by exec. Under Windows, processes are created using the CreateProcess API.

The second workload is a parallel compile of the bash shell sources using the command “make -j 20” in a clean object directory. A compilation workload was chosen because it creates a large number of short-lived processes, stressing Antfarm’s ability to track many concurrent processes that have varying runtimes.

Antfarm incurs no false negatives in any of the tested

	Process Create	Addr Spc Create	Inferred Create	Process Exit	Addr Spc Exit	Inferred Exit	Context Switch	CS Inferred
Linux 2.4 x86								
Fork Only	1000	1000	1000	1000	1000	1000	3331	3331
Fork + Exec	1000	1000	1000	1000	1000	1000	3332	3332
Vfork + Exec	1000	1000	1000	1000	1000	1000	3937	3937
Compile	815	815	815	815	815	815	4447	4447
Linux 2.6 x86								
Fork Only	1000	1000	1000	1000	1000	1000	3939	3939
Fork+Exec	1000	2000	2000	1000	2000	2000	4938	4938
Vfork + Exec	1000	1000	1000	1000	1000	1000	3957	3957
Compile	748	1191	1191	748	1191	1191	2550	2550
Windows								
Create	1000	1000	1000	1000	1000	1000	74431	74431
Compile	2602	2602	2602	2602	2602	2602	835248	835248

Table 2: **Completeness.** The table shows the total number of creations and exits for processes and address spaces reported by the operating system. The total number of process creations and exits inferred by Antfarm are shown in comparison. Antfarm detects all process creates and exits without false positives or false negatives on both Linux 2.4 and Windows. Fork and exec, however, lead to false positives under Linux 2.6 (**bold face values**). All false positives are due to the mismatch between address spaces and processes indicated by matching counts for address space creates and inferred creates. Actual and inferred context switch counts are also shown for completeness and are accurate as expected.

cases, *i.e.*, all process-related events reported by our instrumented OSes are detected by the VMM. The fact that inferred counts are always greater than or equal to the reported counts suggests this, but we also verified that each OS-reported event is properly matched by at least one VMM-inferred event.

Under Linux 2.4 and Windows, no false positives occur, indicating Antfarm can precisely detect address space events and that there is a one-to-one match between address spaces and processes for these operating systems. Under Linux 2.6, however, false positives do occur, indicated in Table 2 by the inferred event counts that are larger than the OS-reported counts. This discrepancy is due to the implementation of the Linux 2.6 fork and exec system calls.

UNIX programs create new user processes by invoking the fork system call which, among other things, constructs a new address space for the child process. The child's address space is a copy of the parent's address space. In most cases, the newly created child process immediately invokes the exec system call which replaces the child's virtual memory image with that of another program read from disk.

In Linux 2.4, when exec is invoked the existing process address space is cleared and reused for the newly loaded program. In contrast, Linux 2.6 destroys and releases the address space of a process invoking exec. A new address space is allocated for the newly exec'd program. Hence, under Linux 2.6, a process that invokes exec has two dis-

tinct address spaces associated with it, which do not overlap in time. In other words, the runtime of the process is *partitioned* into two segments. One segment corresponds to the period between fork and exec and the other corresponds to the period between exec and process exit. Antfarm, because it is based on address space tracking, concludes that two different processes are created leading to twice as many inferred process creations and exits as actually occurred.

Due to the idiomatic use of fork and exec, however, a process is partitioned in a distinctive way. The Linux 2.6/x86 case in Figure 1 depicts the temporal relationship between the two inferred pseudo-processes. The duration of the first pseudo-process will nearly always be small. For example, in the case of our compilation workload, the average time between fork and exec is less than 1 ms, compared to the average lifetime of the second pseudo-process, which is more than 2 seconds, a difference of three orders of magnitude.

The two pseudo-processes are separated by a short time period where neither is active. This interval corresponds to the time after the original address space is destroyed and before the new address space is created. During the compilation workload this interval averaged less than 0.1 ms and was never larger than 2.3 ms. Since no user instructions can be executed in the absence of a user address space, the combination of the two pseudo-processes detected by Antfarm encompasses all user activity of the true process. Conventional use of fork and exec imply

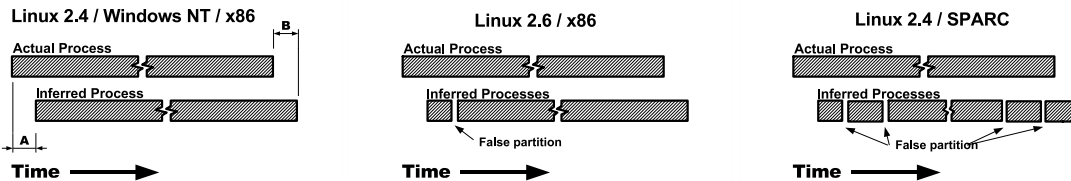


Figure 1: **Effects of error.** The figure shows where each type of process identification error occurs for each tested platform. Error is either lag between when the true event occurs and when the VMM detects it, (e.g., A and B in the figure) or consists of falsely partitioning a single OS process into multiple inferred processes. In Linux 2.6/x86, this only occurs on `exec`, which typically happens immediately after `fork`. On SPARC this partitioning happens whenever a process calls either `fork` or `exec`.

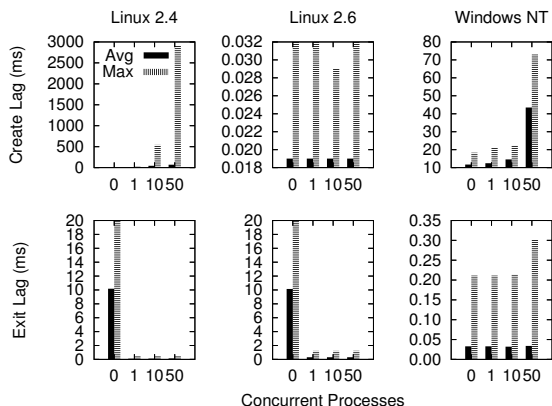


Figure 2: **Lag vs. System Load.** The figure shows average and maximum create and exit lag time measurements for a variety of system load levels in each of our x86 evaluation environments. Average and worst case create lag are affected by system load in Linux 2.4 and Windows, but are small and nearly constant under Linux 2.6. Except for a large exit lag with no competing processes on Linux, exit lag does not appear to be sensitive to system load.

that nearly all substantive activity of the true user process is captured within the second pseudo-process.

6.1.2 Lag

The second aspect of process identification accuracy that we consider is the time difference between a process event and when the same event is detected by the VMM. We define a process to exist at the instant the `fork` (or its equivalent) system call is invoked. Exit is defined as the start of the `exit` system call. These definitions are maximally conservative. In Figure 1 create lag is labeled A and exit lag is labeled B.

Lag is similar in nature to response time, so we expect it to be sensitive to system load. To evaluate this sensitivity, we conduct an experiment that measures lag times for various levels of system load on Linux 2.4, Linux 2.6, and Windows. In each experiment, 0, 1, 10, or 50 CPU-

bound processes were created. 100 additional test processes were then created and the create and exit lag time of each were computed. Test process creations were separated by 10 ms and each test process slept for one second before exiting.

The results of these experiments are presented in Figure 2. For each graph, the x-axis shows the number of concurrent CPU-bound processes and the y-axis shows lag time. Create lag is sensitive to system load on both Linux 2.4 and Windows, as indicated by the steadily increasing lag time for increasing system load. This result is intuitive since a call to the scheduler is likely to occur between the invocation of the create process API in the parent (when a process begins) and when the child process actually runs (when the VMM detects it). Linux 2.6, however, exhibits a different process creation policy that leads to relatively small and constant creation lag. Since Antfarm detects a process creation when a process first runs, the VMM will always be informed of a process's existence before any user instructions are executed.

Exit lag is typically small for each of the platforms. The exception is for an otherwise idle Linux which shows a relatively large exit lag average of 10 ms. The reason for this anomaly is that most Linux kernel tasks, including the idle task, do not need an associated user address space and therefore borrow the previously active user address space when they need to run. This mechanism allows a kernel task to run without incurring the expense of a TLB flush. In the case of this experiment, test processes were started at intervals of 10 ms and each process sleeps for one second; hence, when no other processes are ready to run, approximately 10 ms elapse between process exit and when another process begins. During this interval, the Linux idle task is active and prevents the previous address space from being released, which leads to the observed delay.

6.1.3 The Big Picture

Figure 3 shows a set of timelines depicting how Antfarm tracks process activity over time for a parallel compilation

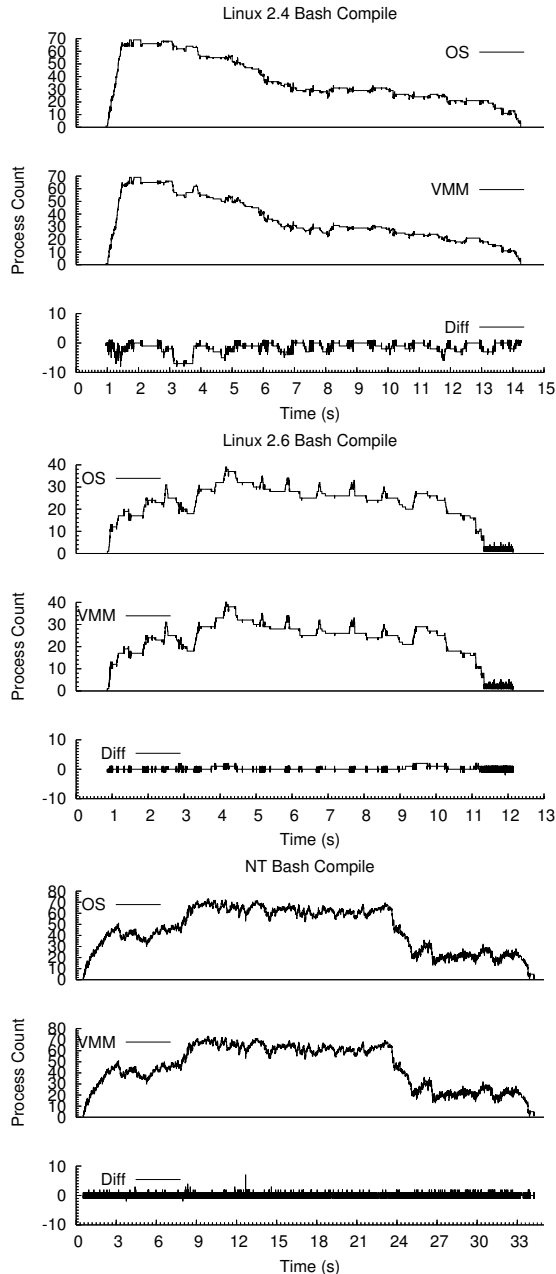


Figure 3: **Compilation Workload Timelines.** For x86/Linux 2.4, x86/Linux 2.6 and x86/Windows a process count timeline is shown. Each timeline depicts the OS-reported process count, the VMM-inferred process count and the difference between the two versus time. Lag has a larger impact on accuracy than false positives. x86/Linux 2.6, which exhibits significantly smaller lag than x86/Linux 2.4 is able to track process counts more accurately.

workload on each of our x86 platforms. The top curve in each graph shows the true, current process count over time as reported by the operating system. The middle curve shows the current process count as inferred by Antfarm. The bottom curve shows the difference between the two curves calculated as $Inferred - Actual$.

The result of the relatively large creation lag under Linux 2.4 is apparent in the larger negative process count differences compared to Linux 2.6. For this workload and metric combination, creation lag is of greater concern than the false positives experienced by Linux 2.6. In another environment such as a more lightly loaded system, which would tend to reduce lag, or for a metric like total cumulative process count, the false positives incurred by Linux 2.6 could be more problematic.

Exit lag is not prominent in any of the graphs. Large, persistent exit lag effects would show up as significant positive deviations in the difference curves. The fact that errors due to fork and exec do not accumulate over time under Linux 2.6 is also apparent because no increasing inaccuracy trend is present.

6.2 Overhead

To evaluate the overhead of our process awareness techniques we measure and compare the runtime of two workloads under Antfarm and under a pristine build of Xen. The first workload is a microbenchmark that represents a worst case performance scenario for Antfarm. Experiments were performed using Linux 2.4 guests.

Since our VMM extensions only affect code paths where page tables are updated, our first microbenchmark focuses execution on those paths. The program allocates 100 MB of memory, touches each page once to ensure a page table entry for every allocated page is created and then exits, causing all of the page tables to be cleared and released. This program is run 100 times and the total elapsed time is computed. The experiment was repeated five times and the average duration is reported. There was negligible variance between experiments. Under an unmodified version of Xen this experiment required an average of 24.75 seconds to complete. Under Antfarm for Xen the experiment took an average of 25.35 seconds to complete. The average slowdown is 2.4% for this worst case example.

The runtime for configuring and building bash was also compared between our modified and unmodified versions of Xen. In the unmodified case the average measured runtime of five trials was 44.49 s. The average runtime of the same experiment under our modified Xen was 44.74 s. The variance between experiments was negligible yielding a slowdown of about 0.6% for this process-intensive application workload.

	Process Create	Addr Spc Create	Inferred Create	Process Exit	Addr Spc Exit	Inferred Exit	Context Switch	CS Inferred
SPARC/Linux								
Fork Only	1000	1000	2000	1000	1000	2000	3419	3419
Fork & Exec	1000	1000	3000	1000	1000	3000	3426	3426
Vfork	1000	1000	1000	1000	1000	1000	4133	4133
Compile	603	603	1396	603	603	1396	1678	1678

Table 3: **Completeness for SPARC.** The table shows the results for the same experiments reported for x86 in Table 2, but for SPARC/Linux 2.4. False positives occur for each fork due to an implementation which uses copy-on-write. Antfarm also infers an additional, non-existent exit/create event pair for each exec. This error is not due to multiple address spaces per process as on x86, but rather stems from the flush that occurs to clear the caller's address space upon exec.

6.3 SPARC Evaluation

Our implementation of process tracking on SPARC uses Simics. Each virtual machine is configured with a 168 MHz UltraSPARC II processor and 256 MB of RAM. We use SPARC/Linux version 2.4.14 as the guest operating system for all tests. The guest operating system is instrumented to report the same information as described for x86.

6.3.1 Completeness

We use the same criteria to evaluate process awareness under SPARC as under x86. Table 3 lists the total event counts for our process creation micro-benchmark and for the bash compilation workload.

As on x86, no false negatives occur. In contrast to x86, the fork-only variant of the microbenchmark incurs false positives. The reason for this is the copy-on-write implementation of fork under Linux. During fork all of the writable portions of the parent's address space are marked read-only so that they can be copy-on-write shared with the child. Many entries in the parent's page tables are updated and all of the corresponding TLB entries must be flushed. SPARC/Linux accomplishes this efficiently by flushing all of the parent's current TLB entries using a context demap operation. The context demap is incorrectly interpreted by Antfarm as a process exit. As soon as the parent is scheduled to run again, we detect the use of the address space and signal a matching spurious process creation.

The false positives caused by the use of fork under SPARC are different in character than those caused by exec under x86. These errors are not limited (by convention) to the usually tiny time interval between fork and exec. They will appear whenever fork is invoked, which for processes like a user shell can occur repeatedly throughout the process's lifetime. The Linux 2.4/SPARC case in Figure 1 depicts how a process that repeatedly

invokes fork might be partitioned into many inferred pseudo-processes by Antfarm.

When exec is used we see additional false positives, but for a different reason than under x86/Linux 2.6. In this case the process inference technique falsely reports the creation of new address spaces that don't really exist. The cause of this behavior is a TLB demap operation that occurs when a process address space is cleared on exec. This error mode is different than under x86 where observed errors were due to a faulty assumption of a single address space per process. On SPARC, the error occurs because our chosen indicator, context demap, can happen without the corresponding address space being deallocated.

Given these two sources of false positives, one would expect our compilation workload to experience approximately the same multiple of false positives as seen for the fork+exec synthetic benchmark. We see, however, fewer false positives than we expect, due to the use of vfork by both GNU make and gcc. Vfork creates a new process but does not duplicate the parent's address space. Since no parent page tables are changed, no flush is required. When exec is invoked we detect the creation of the *single* new address space. Hence, when vfork and exec are used to create new processes under SPARC/Linux, Antfarm experiences no false positives. The build process, however, consists of more than processes created by make and gcc. Many processes are created by calls to an external shell and these process creations induce the false positives we observe.

6.3.2 Lag

Lag between OS-recorded and VMM-inferred process events under SPARC/Linux is comparable to Linux on x86. The average and maximum lag values for SPARC/Linux under various system loads are shown in Figure 4. Create lag is sensitive to system load. Exit lag is unaffected by load as on x86.

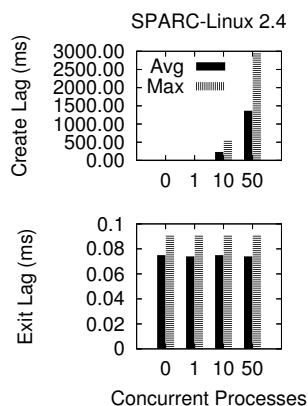


Figure 4: **Lag vs. System Load, SPARC.** The figure shows average and maximum create and exit lag time measurements for the same experiments described in Figure 2. Create lag grows with system load. Exit lag is small and nearly constant, independent of load.

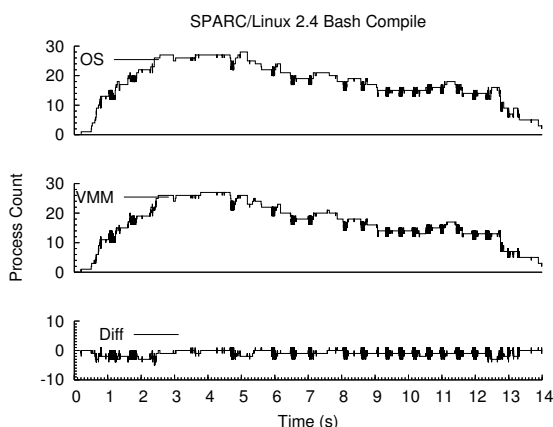


Figure 5: **Compilation Workload Timeline.** Compilation timeline comparable to Figure 3 for SPARC/Linux.

6.3.3 Limitations

While the SPARC inference technique is simple, it suffers drawbacks relative to x86. As shown, the technique incurs more false positives than the x86 techniques. In spite of the additional false positives, Figure 5 shows that the technique can track process events during a parallel compilation workload at least as accurately as x86/Linux 2.4.

Unlike the x86, where one can reasonably assume that a page directory page would not be shared by multiple runnable processes, one cannot make such an assumption for context IDs on SPARC. The reason is the vastly smaller space of unique context IDs. The SPARC provides only 13 bits for this field which allows up to 8192 distinct contexts to be represented concurrently. If a system exceeds this number of active processes, context IDs must necessarily be recycled. In some cases, system soft-

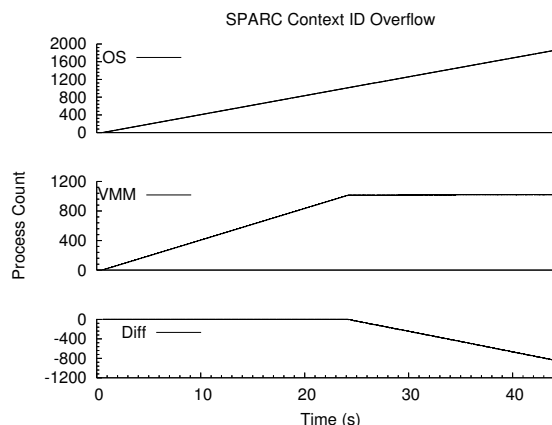


Figure 6: **Context ID Overflow.** When more processes exist than can be represented by the available SPARC context IDs our techniques fail to detect context ID reuse.

ware will further limit the number of concurrent contexts it supports. For example, Linux on SPARC architectures uses only 10 of the available 13 context bits, so only 1024 concurrent address spaces are supported without recycling.

Figure 6 shows the behavior of our SPARC process detection techniques when more processes exist than can be distinguished by the available context IDs. Once the limit is reached at 1024, the technique fails to detect additional process creations.

The importance of this second limitation is somewhat reduced because even very busy servers rarely have more than 1000 active processes, a fact which no doubt influenced the selection of the context ID field's size.

6.4 Discussion

The process event detection techniques used by Antfarm are based on the mechanisms provided by a CPU architecture to implement and manage virtual address spaces, and on the responsibilities of general-purpose operating systems to maintain process isolation. The techniques assume an OS will follow the address space conventions suggested by the MMU features available in an architecture. If an OS deviates from the convention, detection accuracy will likely differ from what we have reported here. Our evaluation shows that two widely used operating systems adhere to our assumptions. Antfarm precisely identifies the desired process events on x86/Windows and x86/Linux 2.4. Some false positives occur under x86/Linux 2.6 and SPARC/Linux. However, the false positives are stylized and affect the ability of Antfarm to keep an accurate process count very little.

New architectures devoted to hardware-assisted virtualization [1, 13] will, in some configurations, reduce or eliminate the need for a VMM to track guest page ta-

ble updates and context switches. For example, AMD plans to include two levels of address translation and a private guest-CR3 as options in its Secure Virtual Machine (SVM) architecture. This fact does not prevent a VMM from observing its guest operating systems; shadow page tables are explicitly supported by these architectures. It will, however, likely increase the performance penalty exacted by the techniques used in Antfarm.

7 Case Study: Anticipatory Scheduling

The order in which disk requests are serviced can make a large difference to disk I/O performance. If requests to adjacent locations on disk are serviced consecutively, the time spent moving the disk head unproductively is minimized. This is the primary performance goal of most disk scheduling algorithms. This case study explores the application of one innovative scheduling algorithm called *anticipatory scheduling* [14] in a virtual machine environment. The implementation makes use of Antfarm for Xen.

7.1 Background

Iyer *et al.* [14] have demonstrated a phenomenon they call *deceptive idleness* for disk access patterns generated by competing processes performing synchronous, sequential reads. Deceptive idleness leads to excessive seeking between locations on disk. Their solution, called anticipatory scheduling, introduces a small amount of waiting time between the completion of one request and the initiation of the next if the process whose disk request just completed is likely to issue another request for a nearby location. This strategy leads to substantial seek savings and throughput gains for concurrent disk access streams that each exhibit spatial locality.

Anticipatory scheduling makes use of process-specific information. It decides whether to wait for a process to issue a new read request and how long to wait based on statistics the disk scheduler keeps for all processes about their recent disk accesses. For example, the average distance from one request to the next is stored as an estimate of how far away the process's next access will be. If this distance is large, there is little sense waiting for the process to issue a request nearby. Statistics about how long a process waits after one request completes before it issues another are also kept in order to determine how long it make sense to wait for the next request to be issued.

Anticipatory scheduling does not work well in a virtual machine environment. System-wide information about disk requests is required to estimate where the disk head is located, which is essential in deciding if a request is

nearby. Information about individual process's I/O behavior is required to determine whether and how long to wait. This information is not completely available to either a single guest, which only knows about its own requests, or to the VMM, which cannot distinguish between guest-level processes. While guests and the VMM could cooperate to implement anticipatory scheduling, this requires the introduction of additional, specialized VMM-to-guest interfaces. New interfaces may not be possible in the case of legacy or binary-only components. In any case, such interfaces do not exist today.

7.2 Information

To implement anticipatory scheduling effectively in a VMM, the VMM must be able to distinguish between guest processes. Additionally, it must be able to associate disk read requests with specific guest processes. Given those two pieces of information, a VMM implementation of anticipatory scheduling can maintain average seek distance and inter-request waiting time for processes across all guests. We use Antfarm to inform an implementation of anticipatory scheduling inside of Xen.

To associate disk read requests to processes, we employ a simple *context association* strategy that associates a read request with whatever process is currently active. This simple strategy does not take potential asynchrony within the operating system into account. For example, due to request queuing inside the OS, a read may be issued to the VMM after the process in which it originated has blocked and context switched off the processor. This leads to association error. We have researched more accurate ways of associating reads to their true originating process by tracking the movement of data from the disk through memory towards the requesting process. These methods have proven effective in overcoming association error due to queuing. Because of limited space, however, we do not present these techniques here. The implementation of anticipatory scheduling described in this paper uses simple context association.

7.3 Implementation

Xen implements I/O using device driver virtual machines (DDVM) [7]. A DDVM is a virtual machine that is allowed unrestricted access to one or more physical devices. DDVMs are logically part of the Xen VMM. Operationally, guests running in normal virtual machines make disk requests to a DDVM via an idealized disk device interface and the DDVM carries out the I/O on their behalf. In current versions of Xen, these driver VMs run Linux to take advantage of the broad device support it offers. A device back-end in the driver VM services requests submitted by an instance of a front-end driver located in all

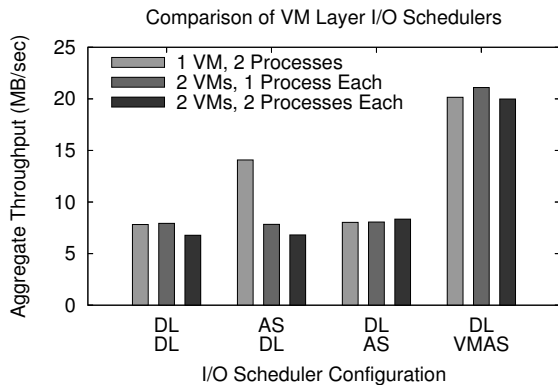


Figure 7: **Benefit of process awareness for anticipatory scheduling.** The graph shows the aggregate throughput for various configurations of I/O scheduler, number of virtual machines and number of processes per virtual machine. The experiment uses the Linux deadline scheduler (DL), the standard anticipatory scheduler (AS), and our VMM-level anticipatory scheduler (VMAS). Adding process awareness enables VMAS to achieve single process sequential read performance in aggregate among competing sequential streams. AS running at the guest layer is somewhat effective in the 1 VM / 2 process case since it has global disk request information.

normal VMs.

The standard Linux kernel includes an implementation of anticipatory scheduling. We implement anticipatory scheduling at the VMM layer by enabling the Linux anticipatory scheduler within a Xen DDVM that manages a disk drive. To make this existing implementation process-aware, we introduce a foreign process abstraction that represents processes running in other VMs. When a disk request arrives from a foreign virtual machine, the Xen back-end queries our process-aware Xen hypervisor about which process is currently active in the foreign virtual machine. Given the ability to distinguish between processes we expect that our VMM-level anticipatory scheduler (VMAS) will improve synchronous read performance for competing processes whether they exist in the same or different VMs.

7.4 Evaluation

To demonstrate the effectiveness of our implementation of VMAS, we repeat one of the experiments from the original anticipatory scheduling paper in a virtual machine environment. Our experiment consists of running multiple instances of a program that sequentially reads a 200 MB segment of a private 1 GB file. We vary the number of processes, the assignment of processes to virtual machines, and the disk scheduler used by guests and by the VMM to explore how process awareness influences the

effectiveness of anticipatory scheduling in a VMM. We make use of the Linux deadline I/O scheduler as our non-anticipatory baseline. Results for each of four scheduler configurations combined with three workloads are shown in Figure 7. The workloads are: (1) one virtual machine with two processes, (2) two virtual machines with one process each, and (3) two virtual machines with two processes each.

The first experiment shows the results from a configuration without anticipatory scheduling. It demonstrates the expected performance when anticipation is not in use for each of the three workloads. On our test system this results in an aggregate throughput of about 8 MB/sec.

The second configuration enables anticipatory scheduling in the guest while the deadline scheduler is used by Xen. In the one virtual machine/two process case, where the guest has complete information about all processes actively reading the disk, we expect that an anticipatory scheduler at the guest level will be effective. The figure shows that this is in fact the case. Anticipatory scheduling is able to improve aggregate throughput by 75% from about 8 MB/sec to about 14 MB/sec. In the other cases guest-level anticipatory scheduling performs about as well as the deadline scheduler due to its lack of information about processes in other virtual machines.

Our third experiment demonstrates the performance of unmodified anticipatory scheduling at the VMM layer. Similar to the case of anticipatory scheduling running at the guest layer we would expect performance improvement for the two-virtual-machine/one-process-each case to be good because a VMM can distinguish between virtual machines just as an operating system can distinguish between processes. The improvement does not occur, however, because of an implementation detail of the Xen DDVM back-end driver. The back-end services all foreign requests in the context of a single dedicated task so the anticipatory scheduler interprets the presented I/O stream as a single process making alternating requests to different parts of the disk. The performance is comparable to the configuration without anticipation for all workloads.

The final configuration shows the benefit of process awareness to anticipatory scheduling implemented at the VMM layer. In each of the workload configurations anticipatory scheduling works well, improving aggregate throughput by more than a factor of two, from about 8 MB/sec to about 20 MB/sec. Because it is implemented at the VMM layer, anticipatory scheduling in this configuration has complete information about all requests reaching the disk. Our process awareness extensions allow it to track statistics for each individual process enabling it to make effective anticipation decisions.

8 Conclusion

The widespread adoption of virtual machines brings with it interesting research opportunities to reevaluate where and how certain operating system services are implemented. Implementing OS-like services in a VMM is made more challenging by the lack of high-level OS and application information.

The techniques developed in this paper and their implementation in Antfarm are an explicit example of how information about one important operating system abstraction, the process, can be accurately and efficiently inferred inside a VMM by observing the interaction of a guest OS with its virtual hardware. This method is a useful alternative to explicitly exporting the required information to the VMM directly. By enabling a VMM to independently infer the information it needs, the VMM is decoupled from the specific vendor, version, and even correctness of the guests it supports.

Acknowledgments

This work is sponsored by the Sandia National Laboratories Doctoral Studies Program, by NSF CCR-0133456, ITR-0325267, CNS-0509474, and by generous donations from Network Appliance and EMC.

References

- [1] AMD. *AMD64 Programmer's Manual, Volume 2: System Programming*. December 2005.
- [2] S. Ballmer. Keynote address. Microsoft Management Summit, April 2005.
- [3] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 1–11, Copper Mountain Resort, Colorado, December 1995.
- [4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, October 1997.
- [5] P. M. Chen and B. D. Noble. When virtual is better than real. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 133. IEEE Computer Society, 2001.
- [6] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.
- [7] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *OASIS ASPLOS 2004 Workshop*, 2004.
- [8] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the USENIX Security Symposium*, pages 103–118, San Diego, CA, USA, August 2004.
- [9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.
- [10] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.
- [11] R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.
- [12] P. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.
- [13] Intel. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*. April 2005.
- [14] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 117–130, Banff, Canada, October 2001.
- [15] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 91–104, Brighton, United Kingdom, October 2005.
- [16] S. T. King and P. M. Chen. Backtracking Intrusions. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.
- [17] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.
- [18] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.
- [19] R. Sekar, T. F. Bowen, and M. E. Segal. On preventing intrusions by process behavior monitoring. In *Proc. Workshop on Intrusion Detection and Network Monitoring*, pages 29–40, Berkeley, CA, USA, 1999. USENIX Association.
- [20] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.
- [21] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, San Diego, California, June 2000.
- [22] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.
- [23] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM '04)*, pages 43–56, San Jose, California, May 2004.
- [24] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.
- [25] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

Optimizing Network Virtualization in Xen

Aravind Menon
EPFL, Switzerland

Alan L. Cox
Rice university, Houston

Willy Zwaenepoel
EPFL, Switzerland

Abstract

In this paper, we propose and evaluate three techniques for optimizing network performance in the Xen virtualized environment. Our techniques retain the basic Xen architecture of locating device drivers in a privileged ‘driver’ domain with access to I/O devices, and providing network access to unprivileged ‘guest’ domains through virtualized network interfaces.

First, we redefine the virtual network interfaces of guest domains to incorporate high-level network offload features available in most modern network cards. We demonstrate the performance benefits of high-level offload functionality in the virtual interface, even when such functionality is not supported in the underlying physical interface. Second, we optimize the implementation of the data transfer path between guest and driver domains. The optimization avoids expensive data remapping operations on the transmit path, and replaces page remapping by data copying on the receive path. Finally, we provide support for guest operating systems to effectively utilize advanced virtual memory features such as superpages and global page mappings.

The overall impact of these optimizations is an improvement in transmit performance of guest domains by a factor of 4.4. The receive performance of the driver domain is improved by 35% and reaches within 7% of native Linux performance. The receive performance in guest domains improves by 18%, but still trails the native Linux performance by 61%. We analyse the performance improvements in detail, and quantify the contribution of each optimization to the overall performance.

1 Introduction

In recent years, there has been a trend towards running network intensive applications, such as Internet servers, in virtual machine (VM) environments, where multiple VMs running on the same machine share the machine’s

network resources. In such an environment, the virtual machine monitor (VMM) virtualizes the machine’s network I/O devices to allow multiple operating systems running in different VMs to access the network concurrently.

Despite the advances in virtualization technology [15, 4], the overhead of network I/O virtualization can still significantly affect the performance of network-intensive applications. For instance, Sugerman et al. [15] report that the CPU utilization required to saturate a 100 Mbps network under Linux 2.2.17 running on VMware Workstation 2.0 was 5 to 6 times higher compared to the utilization under native Linux 2.2.17. Even in the paravirtualized Xen 2.0 VMM [4], Menon et al. [10] report significantly lower network performance under a Linux 2.6.10 guest domain, compared to native Linux performance. They report performance degradation by a factor of 2 to 3x for receive workloads, and a factor of 5x degradation for transmit workloads. The latter study, unlike previous reported results on Xen [4, 5], reports network performance under configurations leading to full CPU saturation.

In this paper, we propose and evaluate a number of optimizations for improving the networking performance under the Xen 2.0 VMM. These optimizations address many of the performance limitations identified by Menon et al. [10]. Starting with version 2.0, the Xen VMM adopted a network I/O architecture that is similar to the hosted virtual machine model [5]. In this architecture, a physical network interface is owned by a special, privileged VM called a *driver domain* that executes the native Linux network interface driver. In contrast, an ordinary VM called a *guest domain* is given access to a virtualized network interface. The virtualized network interface has a front-end device in the guest domain and a back-end device in the corresponding driver domain. The front-end and back-end devices transfer network packets between their domains over an *I/O channel* that is provided by the Xen VMM. Within the driver domain, either Eth-

ernet bridging or IP routing is used to demultiplex incoming network packets and to multiplex outgoing network packets between the physical network interface and the guest domain through its corresponding back-end device.

Our optimizations fall into the following three categories:

1. We add three capabilities to the virtualized network interface: scatter/gather I/O, TCP/IP checksum offload, and TCP segmentation offload (TSO). Scatter/gather I/O and checksum offload improve performance in the guest domain. Scatter/gather I/O eliminates the need for data copying by the Linux implementation of `sendfile()`. TSO improves performance throughout the system. In addition to its well-known effects on TCP performance [11, 9] benefiting the guest domain, it improves performance in the Xen VMM and driver domain by reducing the number of network packets that they must handle.
2. We introduce a faster I/O channel for transferring network packets between the guest and driver domains. The optimizations include transmit mechanisms that avoid a data remap or copy in the common case, and a receive mechanism optimized for small data receives.
3. We present VMM support mechanisms for allowing the use of efficient virtual memory primitives in guest operating systems. These mechanisms allow guest OSes to make use of superpage and global page mappings on the Intel x86 architecture, which significantly reduce the number of TLB misses by guest domains.

Overall, our optimizations improve the transmit performance in guest domains by a factor 4.4. Receive side network performance is improved by 35% for driver domains, and by 18% for guest domains. We also present a detailed breakdown of the performance benefits resulting from each individual optimization. Our evaluation demonstrates the performance benefits of TSO support in the virtual network interface even in the absence of TSO support in the physical network interface. In other words, emulating TSO in software in the driver domain results in higher network performance than performing TCP segmentation in the guest domain.

The outline for the rest of the paper is as follows. In section 2, we describe the Xen network I/O architecture and a summary of its performance overheads as described in previous research. In section 3, we describe the design of the new virtual interface architecture. In section 4, we describe our optimizations to the I/O channel. Section 5 describes the new virtual memory optimization mechanisms added to Xen. Section 6 presents

an evaluation of the different optimizations described. In section 7, we discuss related work, and we conclude in section 8.

2 Background

The network I/O virtualization architecture in Xen can be a significant source of overhead for networking performance in guest domains [10]. In this section, we describe the overheads associated with different aspects of the Xen network I/O architecture, and their overall impact on guest network performance. To understand these overheads better, we first describe the network virtualization architecture used in Xen.

The Xen VMM uses an I/O architecture which is similar to the hosted VMM architecture [5]. Privileged domains, called ‘driver’ domains, use their native device drivers to access I/O devices directly, and perform I/O operations on behalf of other unprivileged domains, called guest domains. Guest domains use virtual I/O devices controlled by paravirtualized drivers to request the driver domain for device access.

The network architecture used in Xen is shown in figure 1. Xen provides each guest domain with a number of virtual network interfaces, which is used by the guest domain for all its network communications. Corresponding to each virtual interface in a guest domain, a ‘back-end’ interface is created in the driver domain, which acts as the proxy for that virtual interface in the driver domain. The virtual and backend interfaces are ‘connected’ to each other over an ‘I/O channel’. The I/O channel implements a zero-copy data transfer mechanism for exchanging packets between the virtual interface and backend interfaces by remapping the physical page containing the packet into the target domain.

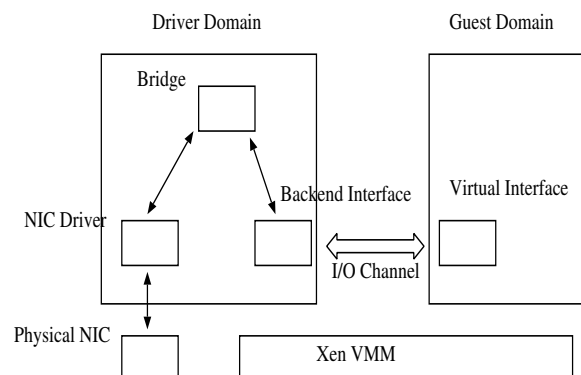


Figure 1: Xen Network I/O Architecture

All the backend interfaces in the driver domain (corresponding to the virtual interfaces) are connected to the physical NIC and to each other through a virtual network

bridge.¹ The combination of the I/O channel and network bridge thus establishes a communication path between the guest's virtual interfaces and the physical interface.

Table 1 compares the transmit and receive bandwidth achieved in a Linux guest domain using this virtual interface architecture, with the performance achieved with the benchmark running in the driver domain and in native Linux. These results are obtained using a netperf-like [1] benchmark which used the zero-copy `sendfile()` for transmit.

Configuration	Receive (Mb/s)	Transmit (Mb/s)
Linux	2508 (100%)	3760 (100%)
Xen driver	1738 (69.3%)	3760 (100%)
Xen guest	820 (32.7%)	750 (19.9%)

Table 1: Network performance under Xen

The driver domain configuration shows performance comparable to native Linux for the transmit case and a degradation of 30% for the receive case. However, this configuration uses native device drivers to directly access the network device, and thus the virtualization overhead is limited to some low-level functions such as interrupts.

In contrast, in the guest domain configuration, which uses virtualized network interfaces, the impact of network I/O virtualization is much more pronounced. The receive performance in guest domains suffers from a performance degradation of 67% relative to native Linux, and the transmit performance achieves only 20% of the throughput achievable under native Linux.

Menon et al. [10] report similar results. Moreover, they introduce Xenoprof, a variant of Oprofile [2] for Xen, and apply it to break down the performance of a similar benchmark. Their study made the following observations about the overheads associated with different aspects of the Xen network virtualization architecture.

Since each packet transmitted or received on a guest domain's virtual interface had to pass through the I/O channel and the network bridge, a significant fraction of the network processing time was spent in the Xen VMM and the driver domain respectively. For instance, 70% of the execution time for receiving a packet in the guest domain was spent in transferring the packet through the driver domain and the Xen VMM from the physical interface. Similarly, 60% of the processing time for a transmit operation was spent in transferring the packet from the guest's virtual interface to the physical interface. The breakdown of this processing overhead was roughly 40% Xen, 30% driver domain for receive traffic, and 30% Xen, 30% driver domain for transmit traffic.

In general, both the guest domains and the driver domain were seen to suffer from a significantly higher TLB

miss rate compared to execution in native Linux. Additionally, guest domains were seen to suffer from much higher L2 cache misses compared to native Linux.

3 Virtual Interface Optimizations

Network I/O is supported in guest domains by providing each guest domain with a set of virtual network interfaces, which are multiplexed onto the physical interfaces using the mechanisms described in section 2. The virtual network interface provides the abstraction of a simple, low-level network interface to the guest domain, which uses paravirtualized drivers to perform I/O on this interface. The network I/O operations supported on the virtual interface consist of simple network transmit and receive operations, which are easily mappable onto corresponding operations on the physical NIC.

Choosing a simple, low-level interface for virtual network interfaces allows the virtualized interface to be easily supported across a large number of physical interfaces available, each with different network processing capabilities. However, this also prevents the virtual interface from taking advantage of different network offload capabilities of the physical NIC, such as checksum offload, scatter/gather DMA support, TCP segmentation offload (TSO).

3.1 Virtual Interface Architecture

We propose a new virtual interface architecture in which the virtual network interface always supports a fixed set of high level network offload features, irrespective of whether these features are supported in the physical network interfaces. The architecture makes use of offload features of the physical NIC itself if they match the offload requirements of the virtual network interface. If the required features are not supported by the physical interface, the new interface architecture provides support for these features in software.

Figure 2 shows the top-level design of the virtual interface architecture. The virtual interface in the guest domain supports a set of high-level offload features, which are reported to the guest OS by the front-end driver controlling the interface. In our implementation, the features supported by the virtual interface are checksum offloading, scatter/gather I/O and TCP segmentation offloading.

Supporting high-level features like scatter/gather I/O and TSO allows the guest domain to transmit network packets in sizes much bigger than the network MTU, and which can consist of multiple fragments. These large, fragmented packets are transferred from the virtual to the physical interface over a modified I/O channel and network bridge (The I/O channel is modified to allow transfer of packets consisting of multiple fragments, and

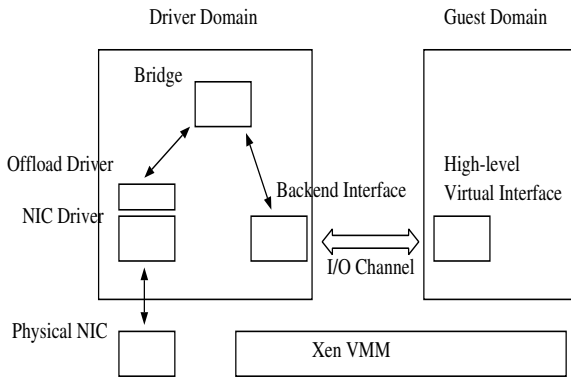


Figure 2: New I/O Architecture

the network bridge is modified to support forwarding of packets larger than the MTU).

The architecture introduces a new component, a ‘software offload’ driver, which sits above the network device driver in the driver domain, and intercepts all packets to be transmitted on the network interface. When the guest domain’s packet arrives at the physical interface, the offload driver determines whether the offload requirements of the packet are compatible with the capabilities of the NIC, and takes different actions accordingly.

If the NIC supports the offload features required by the packet, the offload driver simply forwards the packet to the network device driver for transmission.

In the absence of support from the physical NIC, the offload driver performs the necessary offload actions in software. Thus, if the NIC does not support TSO, the offload driver segments the large packet into appropriate MTU sized packets before sending them for transmission. Similarly, it takes care of the other offload requirements, scatter/gather I/O and checksum offloading if these are not supported by the NIC.

3.2 Advantage of a high level interface

A high level virtual network interface can reduce the network processing overhead in the guest domain by allowing it to offload some network processing to the physical NIC. Support for scatter/gather I/O is especially useful for doing zero-copy network transmits, such as sendfile in Linux. Gather I/O allows the OS to construct network packets consisting of multiple fragments directly from the file system buffers, without having to first copy them out to a contiguous location in memory. Support for TSO allows the OS to do transmit processing on network packets of size much larger than the MTU, thus reducing the per-byte processing overhead by requiring fewer packets to transmit the same amount of data.

Apart from its benefits for the guest domain, a signif-

icant advantage of high level interfaces comes from its impact on improving the efficiency of the network virtualization architecture in Xen.

As described in section 2, roughly 60% of the execution time for a transmit operation from a guest domain is spent in the VMM and driver domain for multiplexing the packet from the virtual to the physical network interface. Most of this overhead is a per-packet overhead incurred in transferring the packet over the I/O channel and the network bridge, and in packet processing overheads at the backend and physical network interfaces.

Using a high level virtual interface improves the efficiency of the virtualization architecture by reducing the number of packets transfers required over the I/O channel and network bridge for transmitting the same amount of data (by using TSO), and thus reducing the per-byte virtualization overhead incurred by the driver domain and the VMM.

For instance, in the absence of support for TSO in the virtual interface, each 1500 (MTU) byte packet transmitted by the guest domain requires one page remap operation over the I/O channel and one forwarding operation over the network bridge. In contrast, if the virtual interface supports TSO, the OS uses much bigger sized packets comprising of multiple pages, and in this case, each remap operation over the I/O channel can potentially transfer 4096 bytes of data. Similarly, with larger packets, much fewer packet forwarding operations are required over the network bridge.

Supporting larger sized packets in the virtual network interface (using TSO) can thus significantly reduce the overheads incurred in network virtualization along the transmit path. It is for this reason that the software offload driver is situated in the driver domain at a point where the transmit packet would have already covered much of the multiplexing path, and is ready for transmission on the physical interface.

Thus, even in the case when the offload driver may have to perform network offloading for the packet in software, we expect the benefits of reduced virtualization overhead along the transmit path to show up in overall performance benefits. Our evaluation in section 6 shows that this is indeed the case.

4 I/O Channel Optimizations

Previous research [10] noted that 30-40% of execution time for a network transmit or receive operation was spent in the Xen VMM, which included the time for page remapping and ownership transfers over the I/O channel and switching overheads between the guest and driver domain.

The I/O channel implements a zero-copy page remapping mechanism for transferring packets between the

guest and driver domain. The physical page containing the packet is remapped into the address space of the target domain. In the case of a receive operation, the ownership of the physical page itself which contains the packet is transferred from the driver to the guest domain (Both these operations require each network packet to be allocated on a separate physical page).

A study of the implementation of the I/O channel in Xen reveals that three address remaps and two memory allocation/deallocation operations are required for each packet receive operation, and two address remaps are required for each packet transmit operation. On the receive path, for each network packet (physical page) transferred from the driver to the guest domain, the guest domain releases ownership of a page to the hypervisor, and the driver domain acquires a replacement page from the hypervisor, to keep the overall memory allocation constant. Further, each page acquire or release operation requires a change in the virtual address mapping. Thus overall, for each receive operation on a virtual interface, two page allocation/freeing operations and three address remapping operations are required. For the transmit path, ownership transfer operation is avoided as the packet can be directly mapped into the privileged driver domain. Each transmit operation thus incurs two address remapping operations.

We now describe alternate mechanisms for packet transfer on the transmit path and the receive path.

4.1 Transmit Path Optimization

We observe that for network transmit operations, the driver domain does not need to map in the entire packet from the guest domain, if the destination of the packet is any host other than the driver domain itself. In order to forward the packet over the network bridge, (from the guest domain's backend interface to its target interface), the driver domain only needs to examine the MAC header of the packet. Thus, if the packet header can be supplied to the driver domain separately, the rest of the network packet does not need to be mapped in.

The network packet needs to be mapped into the driver domain only when the destination of the packet is the driver domain itself, or when it is a broadcast packet.

We use this observation to avoid the page remapping operation over the I/O channel in the common transmit case. We augment the I/O channel with an out-of-band 'header' channel, which the guest domain uses to supply the header of the packet to be transmitted to the backend driver. The backend driver reads the header of the packet from this channel to determine if it needs to map in the entire packet (i.e., if the destination of the packet is the driver domain itself or the broadcast address). It then constructs a network packet from the packet header and the (possibly unmapped) packet fragments, and forwards

this over the network bridge.

To ensure correct execution of the network driver (or the backend driver) for this packet, the backend ensures that the pseudo-physical to physical mappings of the packet fragments are set correctly (When a foreign page frame is mapped into the driver domain, the pseudo-physical to physical mappings for the page have to be updated). Since the network driver uses only the physical addresses of the packet fragments, and not their virtual address, it is safe to pass an unmapped page fragment to the driver.

The 'header' channel for transferring the packet header is implemented using a separate set of shared pages between the guest and driver domain, for each virtual interface of the guest domain. With this mechanism, the cost of two page remaps per transmit operation is replaced in the common case, by the cost of copying a small header.

We note that this mechanism requires the physical network interface to support gather DMA, since the transmitted packet consists of a packet header and unmapped fragments. If the NIC does not support gather DMA, the entire packet needs to be mapped in before it can be copied out to a contiguous memory location. This check is performed by the software offload driver (described in the previous section), which performs the remap operation if necessary.

4.2 Receive Path Optimization

The Xen I/O channel uses page remapping on the receive path to avoid the cost of an extra data copy. Previous research [13] has also shown that avoiding data copy in network operations significantly improves system performance.

However, we note that the I/O channel mechanism can incur significant overhead if the size of the packet transferred is very small, for example a few hundred bytes. In this case, it may not be worthwhile to avoid the data copy.

Further, data transfer by page transfer from the driver domain to the guest domain incurs some additional overheads. Firstly, each network packet has to be allocated on a separate page, so that it can be remapped. Additionally, the driver domain has to ensure that there is no potential leakage of information by the remapping of a page from the driver to a guest domain. The driver domain ensures this by zeroing out at initialization, all pages which can be potentially transferred out to guest domains. (The zeroing is done whenever new pages are added to the memory allocator in the driver domain used for allocating network socket buffers).

We investigate the alternate mechanism, in which packets are transferred to the guest domain by data copy.

As our evaluation shows, packet transfer by data copy instead of page remapping in fact results in a small improvement in the receiver performance in the guest domain. In addition, using copying instead of remapping allows us to use regular MTU sized buffers for network packets, which avoids the overheads incurred from the need to zero out the pages in the driver domain.

The data copy in the I/O channel is implemented using a set of shared pages between the guest and driver domains, which is established at setup time. A single set of pages is shared for all the virtual network interfaces in the guest domain (in order to restrict the copying overhead for a large working set size). Only one extra data copy is involved from the driver domain's network packets to the shared memory pool.

Sharing memory pages between the guest and the driver domain (for both the receive path, and for the header channel in the transmit path) does not introduce any new vulnerability for the driver domain. The guest domain cannot crash the driver domain by doing invalid writes to the shared memory pool since, on the receive path, the driver domain does not read from the shared pool, and on the transmit path, it would need to read the packet headers from the guest domain even in the original I/O channel implementation.

5 Virtual Memory Optimizations

It was noted in a previous study [10], that guest operating systems running on Xen (both driver and guest domains) incurred a significantly higher number of TLB misses for network workloads (more than an order of magnitude higher) relative to the TLB misses in native Linux execution. It was conjectured that this was due to the increase in working set size when running on Xen.

We show that it is the absence of support for certain virtual memory primitives, such as superpage mappings and global page table mappings, that leads to a marked increase in TLB miss rate for guest OSes running on Xen.

5.1 Virtual Memory features

Superpage and global page mappings are features introduced in the Intel x86 processor series starting with the Pentium and Pentium Pro processors respectively.

A superpage mapping allows the operating system to specify the virtual address translation for a large set of pages, instead of at the granularity of individual pages, as with regular paging. A superpage page table entry provides address translation from a set of contiguous virtual address pages to a set of contiguous physical address pages.

On the x86 platform, one superpage entry covers 1024 pages of physical memory, which greatly increases the virtual memory coverage in the TLB. Thus, this greatly reduces the capacity misses incurred in the TLB. Many operating systems use superpages to improve their overall TLB performance: Linux uses superpages to map the 'linear' (lowmem) part of the kernel address space; FreeBSD supports superpage mappings for both user-space and kernel space translations [12].

The support for global page table mappings in the processor allows certain page table entries to be marked 'global', which are then kept persistent in the TLB across TLB flushes (for example, on context switch). Linux uses global page mappings to map the kernel part of the address space of each process, since this part is common between all processes. By doing so, the kernel address mappings are not flushed from the TLB when the processor switches from one process to another.

5.2 Issues in supporting VM features

5.2.1 Superpage Mappings

A superpage mapping maps a contiguous virtual address range to a contiguous physical address range. Thus, in order to use a superpage mapping, the guest OS must be able to determine physical contiguity of its page frames within a superpage block. This is not possible in a fully virtualized system like VMware ESX server [16], where the guest OS uses contiguous 'pseudo-physical' addresses, which are transparently mapped to discontinuous physical addresses.

A second issue with the use of superpages is the fact that all page frames within a superpage block must have identical memory protection permissions. This can be problematic in a VM environment because the VMM may want to set special protection bits for certain page frames. As an example, page frames containing page tables of the guest OS must be set read-only so that the guest OS cannot modify them without notifying the VMM. Similarly, the GDT and LDT pages on the x86 architecture must be set read-only.

These special permission pages interfere with the use of superpages. A superpage block containing such special pages must use regular granularity paging to support different permissions for different constituent pages.

5.2.2 Global Mappings

Xen does not allow guest OSes to use global mappings since it needs to fully flush the TLB when switching between domains. Xen itself uses global page mappings to map its address range.

5.3 Support for Virtual Memory primitives in Xen

5.3.1 Superpage Mappings

In the Xen VMM, supporting superpages for guest OSes is simplified because of the use of the paravirtualization approach. In the Xen approach, the guest OS is already aware of the physical layout of its memory pages. The Xen VMM provides the guest OS with a pseudo-physical to physical page translation table, which can be used by the guest OS to determine the physical contiguity of pages.

To allow the use of superpage mappings in the guest OS, we modify the Xen VMM and the guest OS to co-operate with each other over the allocation of physical memory and the use of superpage mappings.

The VMM is modified so that for each guest OS, it tries to give the OS an initial memory allocation such that page frames within a superpage block are also physically contiguous (Basically, the VMM tries to allocate memory to the guest OS in chunks of superpage size, i.e., 4 MB). Since this is not always possible, the VMM does not guarantee that page allocation for each superpage range is physically contiguous. The guest OS is modified so that it uses superpage mappings for a virtual address range only if it determines that the underlying set of physical pages is also contiguous.

As noted above in section 5.2.1, the use of pages with restricted permissions, such as pagetable (PT) pages, prevents the guest OS from using a superpage to cover the physical pages in that address range.

As new processes are created in the system, the OS frequently needs to allocate (read-only) pages for the process's page tables. Each such allocation of a read-only page potentially forces the OS to convert the superpage mapping covering that page to a two-level regular page mapping. With the proliferation of such read-only pages over time, the OS would end up using regular paging to address much of its address space.

The basic problem here is that the PT page frames for new processes are allocated randomly from all over memory, without any locality, thus breaking multiple superpage mappings. We solve this problem by using a special memory allocator in the guest OS for allocating page frames with restricted permissions. This allocator tries to group together all memory pages with restricted permissions into a contiguous range within a superpage.

When the guest OS allocates a PT frame from this allocator, the allocator reserves the entire superpage containing this PT frame for future use. It then marks the entire superpage as read-only, and reserves the pages of this superpage for read-only use. On subsequent requests for PT frames from the guest OS, the allocator returns pages from the reserved set of pages. PT page frames freed by

the guest OS are returned to this reserved pool. Thus, this mechanism collects all pages with the same permission into a different superpage, and avoids the breakup of superpages into regular pages.

Certain read-only pages in the Linux guest OS, such as the boot time GDT, LDT, initial page table (`init_mm`), are currently allocated at static locations in the OS binary. In order to allow the use of superpages over the entire kernel address range, the guest OS is modified to relocate these read-only pages to within a read-only superpage allocated by the special allocator.

5.3.2 Global Page Mappings

Supporting global page mappings for guest domains running in a VM environment is quite simple on the x86 architecture. The x86 architecture allows some mechanisms by which global page mappings can be invalidated from the TLB (This can be done, for example, by disabling and re-enabling the global paging flag in the processor control registers, specifically the PGE flag in the CR4 register).

We modify the Xen VMM to allow guest OSes to use global page mappings in their address space. On each domain switch, the VMM is modified to flush all TLB entries, using the mechanism described above. This has the additional side effect that the VMM's global page mappings are also invalidated on a domain switch.

The use of global page mappings potentially improves the TLB performance in the absence of domain switches. However, in the case when multiple domains have to be switched frequently, the benefits of global mappings may be much reduced. In this case, use of global mappings in the guest OS forces both the guest's and the VMM's TLB mappings to be invalidated on each switch. Our evaluation in section 6 shows that global mappings are beneficial only when there is a single driver domain, and not in the presence of guest domains.

We make one additional optimization to the domain switching code in the VMM. Currently, the VMM flushes the TLB whenever it switches to a non-idle domain. We notice that this incurs an unnecessary TLB flush when the VMM switches from a domain *d* to the idle domain and then back to the domain *d*. We make a modification to avoid the unnecessary flush in this case.

5.4 Outstanding issues

There remain a few aspects of virtualization which are difficult to reconcile with the use of superpages in the guest OS. We briefly mention them here.

5.4.1 Transparent page sharing

Transparent page sharing between virtual machines is an effective mechanism to reduce the memory usage in the system when there are a large number of VMs [16]. Page sharing uses address translation from pseudo-physical to physical addresses to transparently share pseudo-physical pages which have the same content.

Page sharing potentially breaks the contiguity of physical page frames. With the use of superpages, either the entire superpage must be shared between the VMs, or no page within the superpage can be shared. This can significantly reduce the scope for memory sharing between VMs.

Although Xen does not make use of page sharing currently, this is a potentially important issue for superpages.

5.4.2 Ballooning driver

The ballooning driver [16] is a mechanism by which the VMM can efficiently vary the memory allocation of guest OSes. Since the use of a ballooning driver potentially breaks the contiguity of physical pages allocated to a guest, this invalidates the use of superpages for that address range.

A possible solution is to force memory allocation/deallocation to be in units of superpage size for coarse grained ballooning operations, and to invalidate superpage mappings only for fine grained ballooning operations. This functionality is not implemented in the current prototype.

6 Evaluation

The optimizations described in the previous sections have been implemented in Xen version 2.0.6, running Linux guest operating systems version 2.6.11.

6.1 Experimental Setup

We use two micro-benchmarks, a transmit and a receive benchmark, to evaluate the networking performance of guest and driver domains. These benchmarks are similar to the netperf [1] TCP streaming benchmark, which measures the maximum TCP streaming throughput over a single TCP connection. Our benchmark is modified to use the zero-copy sendfile system call for transmit operations.

The ‘server’ system for running the benchmark is a Dell PowerEdge 1600 SC, 2.4 GHz Intel Xeon machine. This machine has four Intel Pro-1000 Gigabit NICs. The ‘clients’ for running an experiment consist of Intel Xeon

machines with a similar CPU configuration, and having one Intel Pro-1000 Gigabit NIC per machine. All the NICs have support for TSO, scatter/gather I/O and checksum offload. The clients and server machines are connected over a Gigabit switch.

The experiments measure the maximum throughput achievable with the benchmark (either transmitter or receiver) running on the server machine. The server is connected to each client machine over a different network interface, and uses one TCP connection per client. We use as many clients as required to saturate the server CPU, and measure the throughput under different Xen and Linux configurations. All the profiling results presented in this section are obtained using the Xenoprof system wide profiler in Xen [10].

6.2 Overall Results

We evaluate the following configurations: ‘Linux’ refers to the baseline unmodified Linux version 2.6.11 running native mode. ‘Xen-driver’ refers to the unoptimized XenoLinux driver domain. ‘Xen-driver-opt’ refers to the Xen driver domain with our optimizations. ‘Xen-guest’ refers to the unoptimized, existing Xen guest domain. ‘Xen-guest-opt’ refers to the optimized version of the guest domain.

Figure 3 compares the transmit throughput achieved under the above 5 configurations. Figure 4 shows receive performance in the different configurations.

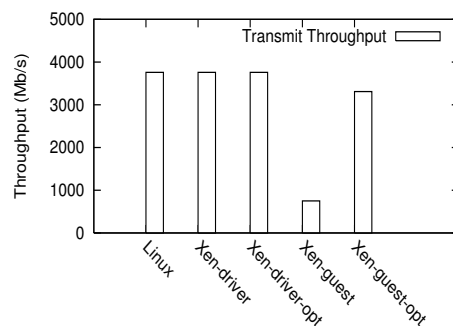


Figure 3: Transmit throughput in different configurations

For the transmit benchmark, the performance of the Linux, Xen-driver and Xen-driver-opt configurations is limited by the network interface bandwidth, and does not fully saturate the CPU. All three configurations achieve an aggregate link speed throughput of 3760 Mb/s. The CPU utilization values for saturating the network in the three configurations are, respectively, 40%, 46% and 43%.

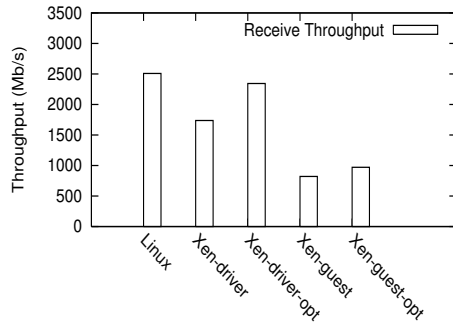


Figure 4: Receive throughput in different configurations

The optimized guest domain configuration, Xen-guest-opt improves on the performance of the unoptimized Xen guest by a factor of 4.4, increasing transmit throughput from 750 Mb/s to 3310 Mb/s. However, Xen-guest-opt gets CPU saturated at this throughput, whereas the Linux configuration reaches a CPU utilization of roughly 40% to saturate 4 NICs.

For the receive benchmark, the unoptimized Xen driver domain configuration, Xen-driver, achieves a throughput of 1738 Mb/s, which is only 69% of the native Linux throughput, 2508 Mb/s. The optimized Xen-driver version, Xen-driver-opt, improves upon this performance by 35%, and achieves a throughput of 2343 Mb/s. The optimized guest configuration Xen-guest-opt improves the guest domain receive performance only slightly, from 820 Mb/s to 970 Mb/s.

6.3 Transmit Workload

We now examine the contribution of individual optimizations for the transmit workload. Figure 5 shows the transmit performance under different combinations of optimizations. ‘Guest-none’ is the guest domain configuration with no optimizations. ‘Guest-sp’ is the guest domain configuration using only the superpage optimization. ‘Guest-ioc’ uses only the I/O channel optimization. ‘Guest-high’ uses only the high level virtual interface optimization. ‘Guest-high-ioc’ uses both high level interfaces and the optimized I/O channel. ‘Guest-high-ioc-sp’ uses all the optimizations: high level interface, I/O channel optimizations and superpages.

The single biggest contribution to guest transmit performance comes from the use of a high-level virtual interface (Guest-high configuration). This optimization improves guest performance by 272%, from 750 Mb/s to 2794 Mb/s.

The I/O channel optimization yields an incremental improvement of 439 Mb/s (15.7%) over the Guest-high

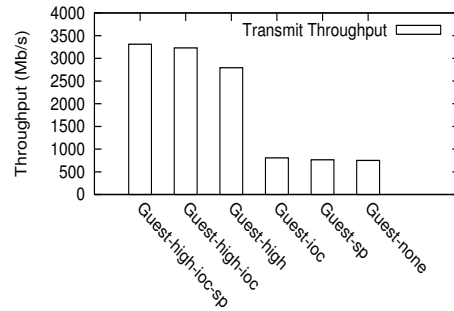


Figure 5: Contribution of individual transmit optimizations

configuration to yield 3230 Mb/s (configuration Guest-high-ioc). The superpage optimization improves this further to achieve 3310 Mb/s (configuration Guest-high-ioc-sp). The impact of these two optimizations by themselves in the absence of the high level interface optimization is insignificant.

6.3.1 High level Interface

We explain the improved performance resulting from the use of a high level interface in figure 6. The figure compares the execution overhead (in millions of CPU cycles per second) of the guest domain, driver domain and the Xen VMM, incurred for running the transmit workload on a single NIC, compared between the high-level Guest-high configuration and the Guest-none configuration.

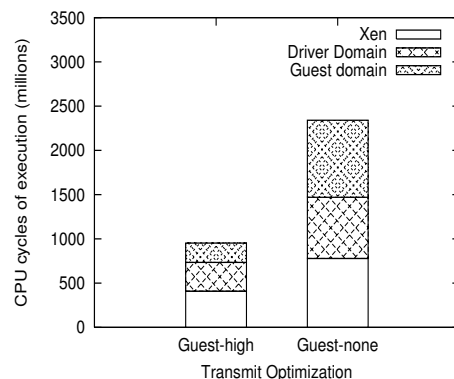


Figure 6: Breakdown of execution cost in high-level and unoptimized virtual interface

The use of the high level virtual interface reduces the execution cost of the guest domain by almost a factor of 4 compared to the execution cost with a low-level interface. Further, the high-level interface also reduces the Xen VMM execution overhead by a factor of 1.9, and

the driver domain overhead by a factor of 2.1. These reductions can be explained by the reasoning given in section 3, namely the absence of data copy because of the support for scatter/gather, and the reduced per-byte processing overhead because of the use of larger packets (TSO).

The use of a high level virtual interface gives performance improvements for the guest domain even when the offload features are not supported in the physical NIC. Figure 7 shows the performance of the guest domain using a high level interface with the physical network interface supporting varying capabilities. The capabilities of the physical NIC form a spectrum, at one end the NIC supports TSO, SG I/O and checksum offload, at the other end it supports no offload feature, with intermediate configurations supporting partial offload capabilities.

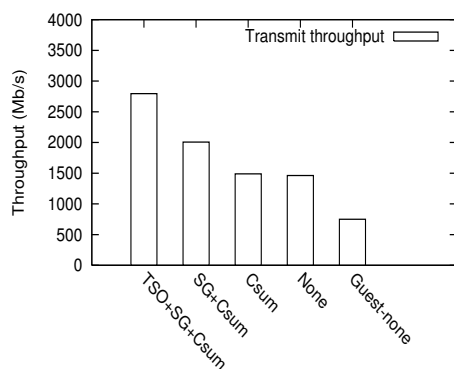


Figure 7: Advantage of higher-level interface

Even in the case when the physical NIC supports no offload feature (bar labeled 'None'), the guest domain with a high level interface performs nearly twice as well as the guest using the default interface (bar labeled Guest-none), viz. 1461 Mb/s vs. 750 Mb/s.

Thus, even in the absence of offload features in the NIC, by performing the offload computation just before transmitting it over the NIC (in the offload driver) instead of performing it in the guest domain, we significantly reduce the overheads incurred in the I/O channel and the network bridge on the packet's transmit path.

The performance of the guest domain with just checksum offload support in the NIC is comparable to the performance without any offload support. This is because, in the absence of support for scatter/gather I/O, data copying both with or without checksum computation incur effectively the same cost. With scatter/gather I/O support, transmit throughput increases to 2007 Mb/s.

6.3.2 I/O Channel Optimizations

Figure 5 shows that using the I/O channel optimizations in conjunction with the high level interface (configuration Guest-high-ioc) improves the transmit performance from 2794 Mb/s to 3239 Mb/s, an improvement of 15.7%.

This can be explained by comparing the execution profile of the two guest configurations, as shown in figure 8. The I/O channel optimization reduces the execution overhead incurred in the Xen VMM by 38% (Guest-high-ioc configuration), and this accounts for the corresponding improvement in transmit throughput.

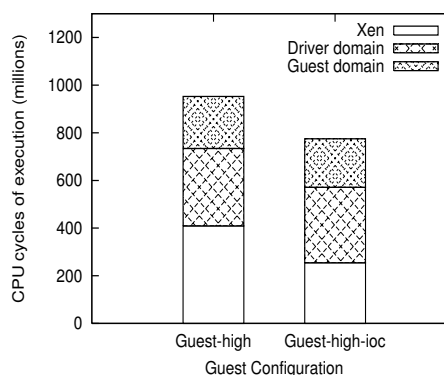


Figure 8: I/O channel optimization benefit

6.3.3 Superpage mappings

Figure 5 shows that superpage mappings improve guest domain performance slightly, by 2.5%. Figure 9 shows data and instruction TLB misses incurred for the transmit benchmark for three sets of virtual memory optimizations. 'Base' refers to a configuration which uses the high-level interface and I/O channel optimizations, but with regular 4K sized pages. 'SP-GP' is the Base configuration with the superpage and global page optimizations in addition. 'SP' is the Base configuration with superpage mappings.

The TLB misses are grouped into three categories: data TLB misses (D-TLB), instruction TLB misses incurred in the guest and driver domains (Guest OS I-TLB), and instruction TLB misses incurred in the Xen VMM (Xen I-TLB).

The use of superpages alone is sufficient to bring down the data TLB misses by a factor of 3.8. The use of global mappings does not have a significant impact on data TLB misses (configurations SP-GP and SP), since frequent switches between the guest and driver domain cancel out any benefits of using global pages.

The use of global mappings, however, does have a negative impact on the instruction TLB misses in the

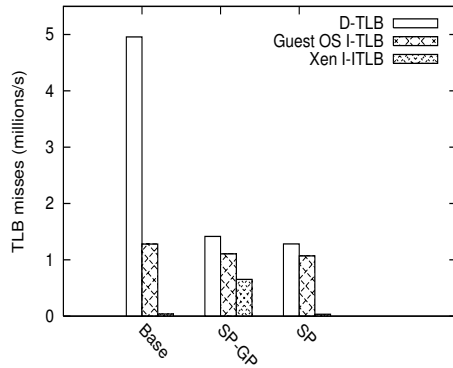


Figure 9: TLB misses for transmit benchmark

Xen VMM. As mentioned in section 5.3.2, the use of global page mappings forces the Xen VMM to flush out all TLB entries, including its own TLB entries, on a domain switch. This shows up as a significant increase in the number of Xen instruction TLB misses. (SP-GP vs. SP).

The overall impact of using global mappings on the transmit performance, however, is not very significant. (Throughput drops from 3310 Mb/s to 3302 Mb/s). The optimal guest domain performance shown in section 6.3 uses only the global page optimization.

6.3.4 Non Zero-copy Transmits

So far we have shown the performance of the guest domain when it uses a zero-copy transmit workload. This workload benefits from the scatter/gather I/O capability in the network interface, which accounts for a significant part of the improvement in performance when using a high level interface.

We now show the performance benefits of using a high level interface, and the other optimizations, when using a benchmark which uses copying writes instead of the zero-copy sendfile. Figure 10 shows the transmit performance of the guest domain for this benchmark under the different combinations of optimizations.

The breakup of the contributions of individual optimizations in this benchmark is similar to that for the sendfile benchmark. The best case transmit performance in this case is 1848 Mb/s, which is much less than the best sendfile throughput (3310 Mb/s), but still significantly better than the unoptimized guest throughput.

6.4 Receive Benchmark

Figure 4 shows that the optimized driver domain configuration, Xen-driver-opt, improves the receive performance from 1738 Mb/s to 2343 Mb/s. The Xen-guest-opt con-

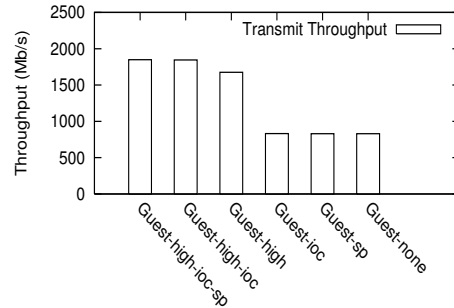


Figure 10: Transmit performance with writes

figuration shows a much smaller improvement in performance, from 820 Mb/s to 970 Mb/s.

6.4.1 Driver domain

We examine the individual contribution of the different optimizations for the receive workload. We evaluate the following configurations: ‘Driver-none’ is the driver domain configuration with no optimizations, ‘Driver-MTU’ is the driver configuration with the I/O channel optimization, which allows the use of MTU sized socket buffers. ‘Driver-MTU-GP’ uses both MTU sized buffers and global page mappings. ‘Driver-MTU-SP’ uses MTU sized buffers with superpages, and ‘Driver-MTU-SP-GP’ uses all three optimizations. ‘Linux’ is the baseline native Linux configuration.

Figure 11 shows the performance of the receive benchmark under the different configurations.

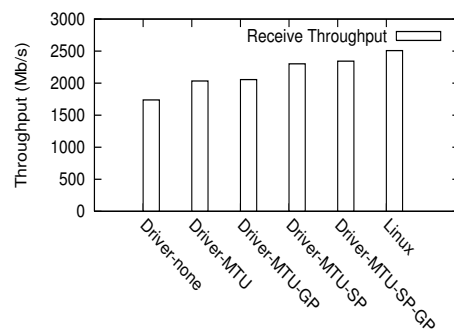


Figure 11: Receive performance in different driver configurations

It is interesting to note that the use of MTU sized socket buffers improves network throughput from 1738 Mb/s to 2033 Mb/s, an increase of 17%. Detailed profiling reveals that the Driver-none configuration, which

uses 4 KB socket buffers, spends a significant fraction of its time zeroing out socket buffer pages. The driver domain needs to zero out socket buffer pages to prevent leakage of information to other domains (section 4.2). The high cost of this operation in the profile indicates that the driver domain is under memory pressure, in which scenario the socket buffer memory allocator constantly frees its buffer pages and then zeroes out newly acquired pages.

The second biggest source of improvement is the use of superpages (configuration Driver-MTU-SP), which improves on the Driver-MTU performance by 13%, from 2033 Mb/s to 2301 Mb/s. The use of global page mappings improves this further to 2343 Mb/s, which is within 7% of the native Linux performance, 2508 Mb/s. The use of global page mappings alone (configuration Driver-MTU-GP) yields an insignificant improvement over the Driver-MTU configuration.

These improvements in performance can be shown to be in direct correspondence with the reduction in TLB misses in the driver domain. Figure 12 shows the effects of the different optimizations on the data TLB misses (millions per sec). For comparison, the data TLB misses incurred in the native Linux configuration is also shown.

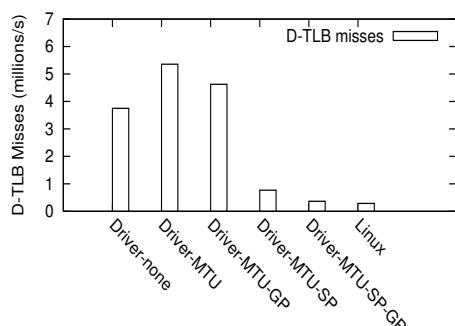


Figure 12: Data TLB misses in Driver domain configurations

The TLB misses incurred under the Xen driver domain configuration (Driver-MTU) are more than an order of magnitude (factor of 18) higher than under the native Linux configuration. The use of superpages in the driver domain (Driver-MTU-SP) eliminates most of the TLB overhead in the driver domain (by 86%). The use of global page mappings (Driver-MTU-GP), by itself, shows only a small improvement in the TLB performance. The combined use of superpages and global mappings brings down the TLB miss count to within 26% of the miss count in Linux.

6.4.2 Guest domain

Figure 13 shows the guest domain performance in three configurations: 'Guest-none' with no optimizations, 'Guest-copy' with the I/O channel optimization and 'Guest-copy-SP' with the I/O channel and superpage optimization. 'Linux' shows the native Linux performance.

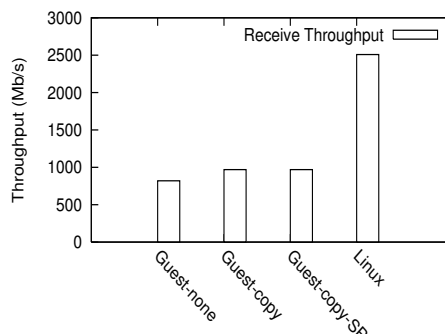


Figure 13: Receive performance in guest domains and Linux

The interesting result from figure 13 is that using copying to implement the I/O channel data transfer between the driver and the guest domain actually performs better than the zero-copy page remapping approach currently used. Copying improves receive performance from 820 Mb/s to 970 Mb/s. As noted in previous sections, the current I/O channel implementation requires the use of two memory allocation/deallocation operations and three page remap operations per packet transfer. The combined cost of these operations is significant enough to outweigh the overhead of data copy for the receive path.

The superpage optimization has no noticeable impact on the receive performance. This is possibly because its benefits are overshadowed by bigger bottlenecks along the receive path. In general, the receive performance in guest domains remains significantly lower than the performance in native Linux. As mentioned in section 2, 70% of the execution time for the receive workload is spent in network virtualization routines in the driver domain and the Xen VMM. Unlike the transmit path optimizations, for the receive path there are no corresponding 'offload' optimizations, which can amortize this cost.

7 Related Work

Virtualization first became available with the IBM VM/370 [6, 14] to allow legacy code to run on new hardware platform. Currently, the most popular full virtu-

alization system is VMware [16]. Several studies have documented the cost of full virtualization, especially on architectures such as the Intel x86.

To address these performance problems, paravirtualization has been introduced, with Xen [4, 5] as its most popular representative. Denali [17] similarly attempts to support a large number of virtual machines.

The use of driver domains to host device drivers has become popular for reasons of reliability and extensibility. Examples include the Xen 2.0 architecture and VMware's hosted workstation [15]. The downside of this approach is a performance penalty for device access, documented, among others, in Sugerman et al. [15] and in Menon et al. [10].

Sugerman et al. describe the VMware hosted virtual machine monitor in [15], and describe the major sources of network virtualization overhead in this architecture. In a fully virtualized system, 'world switches' incurred for emulating I/O access to virtual devices are the biggest source of overhead. The paper describes mechanisms to reduce the number of world switches to improve network performance.

King et al. [8] describe optimizations to improve overall system performance for Type-II virtual machines. Their optimizations include reducing context switches to the VMM process by moving the VM support into the host kernel, reducing memory protection overhead by using segment bounds, and reducing context switch overhead by supporting multiple address spaces within a single process.

Our work introduces and evaluates mechanisms for improving network performance in the paravirtualized Xen VMM. Some of these mechanisms, such as the high level virtual interface optimization, are general enough to be applicable to other classes of virtual machines as well.

Moving functionality from the host to the network card is a well-known technique. Scatter-gather DMA, TCP checksum offloading, and TCP segmentation offloading [11, 9] are present on high-end network devices. We instead add these optimizations to the virtual interface definition for use by guest domains, and demonstrate the advantages of doing so.

The cost of copying and frequent remapping is well known to the operating system's community, and much work has been done on avoiding costly copies or remap operations (e.g., in IOLite [13]). Our I/O channel optimizations avoid a remap for transmission, and replace a remap by a copy for reception.

The advantages of superpages are also well documented. They are used in many operating systems, for instance in Linux and in FreeBSD [12]. We provide primitives in the VMM to allow these operating systems to use superpages when they run as guest operating systems on

top of the VMM.

Upcoming processor support for virtualization [3, 7] can address the problems associated with flushing global page mappings. Using Xen on a processor that has a tagged TLB can improve performance. A tagged TLB enables attaching address space identifier (ASID) to the TLB entries. With this feature, there is no need to flush the TLB when the processor switches between the hypervisor and the guest OSes, and this reduces the cost of memory operations.

8 Conclusions

In this paper, we presented a number of optimizations to the Xen network virtualization architecture to address network performance problems identified in guest domains.

We add three new capabilities to virtualized network interfaces, TCP segmentation offloading, scatter-gather I/O and TCP checksum offloading, which allow guest domains to take advantage of the corresponding offload features in physical NICs. Equally important, these capabilities also improve the efficiency of the virtualization path connecting the virtual and physical network interfaces.

Our second optimization streamlines the data transfer mechanism between guest and driver domains. We avoid a remap operation in the transmit path, and we replace a remap by a copy in the receive path. Finally, we provide a new memory allocator in the VMM which tries to allocate physically contiguous memory to the guest OS, and thereby allows the guest OS to take advantage of superpages.

Of the three optimizations, the high-level virtual interface contributes the most towards improving transmit performance. The optimized I/O channel and superpage optimizations provide additional incremental benefits. Receive performance in the driver domain benefits from the use of superpages.

Overall, the optimizations improve the transmit throughput of guest domains by a factor of 4.4, and the receive throughput in the driver domain by 35%. The receive performance of guest domains remains a significant bottleneck which remains to be solved.

References

- [1] The netperf benchmark. <http://www.netperf.org/netperf/NetperfPage.html>.
- [2] Oprofile. <http://oprofile.sourceforge.net>.

- [3] Advanced Micro Devices. *Secure Virtual Machine Architecture Reference Manual*, May 2005. Revision 3.01.
- [4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, Oct 2003.
- [5] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, Oct 2004.
- [6] P. H. Gum. System/370 extended architecture: facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, Nov. 1983.
- [7] Intel. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, April 2005.
- [8] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *USENIX Annual Technical Conference*, Jun 2003.
- [9] S. Makineni and R. Iyer. Architectural characterization of TCP/IP packet processing on the Pentium M microprocessor. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004.
- [10] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *First ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, June 2005.
- [11] D. Minturn, G. Regnier, J. Krueger, R. Iyer, and S. Makineni. Addressing TCP/IP Processing Challenges Using the IA and IXP Processors. *Intel Technology Journal*, Nov. 2003.
- [12] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002.
- [13] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, Feb. 2000.
- [14] L. Seawright and R. MacKinnon. Vm/370 - a study of multiplicity and usefulness. *IBM Systems Journal*, pages 44–55, 1979.
- [15] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, Jun 2001.
- [16] C. Waldspurger. Memory resource management in VMware ESX server. In *Operating Systems Design and Implementation (OSDI)*, Dec 2002.
- [17] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali isolation kernel. In *Operating Systems Design and Implementation (OSDI)*, Dec 2002.

Notes

¹Xen also allows IP routing based or NAT based solutions. However, bridging is the most widely used architecture.

High Performance VMM-Bypass I/O in Virtual Machines

Jiuxing Liu[†]

Wei Huang[‡]

Bulent Abali[†]

Dhabaleswar K. Panda[‡]

[†] *IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
{jl, abali}@us.ibm.com*

[‡] *Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{huanwei, panda}@cse.ohio-state.edu*

Abstract

Currently, I/O device virtualization models in virtual machine (VM) environments require involvement of a virtual machine monitor (VMM) and/or a privileged VM for each I/O operation, which may turn out to be a performance bottleneck for systems with high I/O demands, especially those equipped with modern high speed interconnects such as InfiniBand.

In this paper, we propose a new device virtualization model called *VMM-bypass I/O*, which extends the idea of OS-bypass originated from user-level communication. Essentially, VMM-bypass allows time-critical I/O operations to be carried out directly in guest VMs without involvement of the VMM and/or a privileged VM. By exploiting the intelligence found in modern high speed network interfaces, VMM-bypass can significantly improve I/O and communication performance for VMs without sacrificing safety or isolation.

To demonstrate the idea of VMM-bypass, we have developed a prototype called Xen-IB, which offers InfiniBand virtualization support in the Xen 3.0 VM environment. Xen-IB runs with current InfiniBand hardware and does not require modifications to existing user-level applications or kernel-level drivers that use InfiniBand. Our performance measurements show that Xen-IB is able to achieve nearly the same raw performance as the original InfiniBand driver running in a non-virtualized environment.

1 Introduction

Virtual machine (VM) technologies were first introduced in the 1960s [14], but are experiencing a resurgence in recent years and becoming more and more attractive to both the industry and the research communities [35]. A key component in a VM environment is the virtual machine monitor (VMM) (also called hypervisor), which is implemented directly on top of hardware and provides virtualized hardware interfaces to VMs. With the help of VMMs, VM technologies allow running many different

virtual machines in a single physical box, with each virtual machine possibly hosting a different operating system. VMs can also provide secure and portable environments to meet the demanding resource requirements of modern computing systems [9].

In VM environments, device I/O access in guest operating systems can be handled in different ways. For instance, in VMware Workstation, device I/O relies on switching back to the host operating system and user-level emulation [37]. In VMware ESX Server, guest VM I/O operations trap into the VMM, which makes direct access to I/O devices [42]. In Xen [11], device I/O follows a split-driver model. Only an isolated device domain (IDD) has access to the hardware using native device drivers. All other virtual machines (guest VMs, or domains) need to pass the I/O requests to the IDD to access the devices. This control transfer between domains needs involvement of the VMM.

In recent years, network interconnects that provide very low latency (less than 5 μ s) and very high bandwidth (multiple Gbps) are emerging. Examples of these high speed interconnects include Virtual Interface Architecture (VIA) [12], InfiniBand [19], Quadrics [34], and Myrinet [25]. Due to their excellent performance, these interconnects have become strong players in areas such as high performance computing (HPC). To achieve high performance, these interconnects usually have intelligent network interface cards (NICs) which can be used to offload a large part of the host communication protocol processing. The intelligence in the NICs also supports user-level communication, which enables safe direct I/O access from user-level processes (OS-bypass I/O) and contributes to reduced latency and CPU overhead.

VM technologies can greatly benefit computing systems built from the aforementioned high speed interconnects by not only simplifying cluster management for these systems, but also offering much cleaner solutions to tasks such as check-pointing and fail-over. Recently, as these high speed interconnects become more and more

commoditized with their cost going down, they are also used for providing remote I/O access in high-end enterprise systems, which increasingly run in virtualized environments. Therefore, it is very important to provide VM support to high-end systems equipped with these high speed interconnects. However, performance and scalability requirements of these systems pose some challenges. In all the VM I/O access approaches mentioned previously, VMMs have to be involved to make sure that I/O accesses are safe and do not compromise integrity of the system. Therefore, current device I/O access in virtual machines requires context switches between the VMM and guest VMs. Thus, I/O access can suffer from longer latency and higher CPU overhead compared to native I/O access in non-virtualized environments. In some cases, the VMM may also become a performance bottleneck which limits I/O performance in guest VMs. In some of the aforementioned approaches (VM Workstation and Xen), a host operating system or another virtual machine is also involved in the I/O access path. Although these approaches can greatly simplify VMM design by moving device drivers out of the VMM, they may lead to even higher I/O access overhead when requiring context switches between the host operating system and the guest VM or two different VMs.

In this paper, we present a *VMM-bypass* approach for I/O access in VM environments. Our approach takes advantages of features found in modern high speed intelligent network interfaces to allow time-critical operations to be carried out directly in guest VMs while still maintaining system integrity and isolation. With this method, we can remove the bottleneck of going through the VMM or a separate VM for many I/O operations and significantly improve communication and I/O performance. The key idea of our VMM-bypass approach is based on the OS-bypass design of modern high speed network interfaces, which allows user processes to access I/O devices directly in a safe way without going through operating systems. OS-bypass was originally proposed by research communities [41, 40, 29, 6, 33] and later adopted by some commercial interconnects such as InfiniBand. Our idea can be regarded as an extension of OS-bypass designs in the context of VM environments.

To demonstrate the idea of VMM-bypass, we have designed and implemented a prototype called *Xen-IB* to provide virtualization support for InfiniBand in Xen. Basically, our implementation presents to each guest VM a para-virtualized InfiniBand device. Our design requires no modification to existing hardware. Also, through a technique called *high-level virtualization*, we allow current user-level applications and kernel-level modules that utilize InfiniBand to run without changes. Our performance results, which includes benchmarks at the basic InfiniBand level as well as evaluation of upper-layer In-

finiBand protocols such as IP over InfiniBand (IPoIB) [1] and MPI [36], demonstrate that performance of our VMM-bypass approach comes close to that in a native, non-virtualized environment. Although our current implementation is for InfiniBand and Xen, the basic VMM-bypass idea and many of our implementation techniques can be readily applied to other high-speed interconnects and other VMMs.

In summary, the main contributions of our work are:

- We proposed the VMM-bypass approach for I/O accesses in VM environments for modern high speed interconnects. Using this approach, many I/O operations can be performed directly without involvement of a VMM or another VM. Thus, I/O performance can be greatly improved.
- Based on the idea of VMM-bypass, we implemented a prototype, *Xen-IB*, to virtualize InfiniBand devices in Xen guest VMs. Our prototype supports running existing InfiniBand applications and kernel modules in guest VMs without any modification.
- We carried out extensive performance evaluation of our prototype. Our results show that performance of our virtualized InfiniBand device is very close to native InfiniBand devices running in a non-virtualized environment.

The rest of the paper is organized as follows: In Section 2, we present background information, including the Xen VM environment and the InfiniBand architecture. In Section 3, we present the basic idea of VMM-bypass I/O. In Section 4, we discuss the detailed design and implementation of our *Xen-IB* prototype. In Section 5, we discuss several related issues and limitations of our current implementation and how they can be addressed in future. Performance evaluation results are given in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8.

2 Background

In this section, we provide background information for our work. In Section 2.1, we describe how I/O device access is handled in several popular VM environments. In Section 2.3, we describe the OS-bypass feature in modern high speed network interfaces. Since our prototype is based on Xen and InfiniBand, we introduce them in Sections 2.2 and 2.4, respectively.

2.1 I/O Device Access in Virtual Machines

In a VM environment, the VMM plays the central role of virtualizing hardware resources such as CPUs, memory, and I/O devices. To maximize performance, the VMM

can let guest VMs access these resources directly whenever possible. Taking CPU virtualization as an example, a guest VM can execute all non-privileged instructions natively in hardware without intervention of the VMM. However, privileged instructions executed in guest VMs will generate a trap into the VMM. The VMM will then take necessary steps to make sure that the execution can continue without compromising system integrity. Since many CPU intensive workloads seldom use privileged instructions (This is especially true for applications in HPC area.), they can achieve excellent performance even when executed in a VM.

I/O device access in VMs, however, is a completely different story. Since I/O devices are usually shared among all VMs in a physical machine, the VMM has to make sure that accesses to them are legal and consistent. Currently, this requires VMM intervention on every I/O access from guest VMs. For example, in VMware ESX Server [42], all physical I/O accesses are carried out within the VMM, which includes device drivers for popular server hardware. System integrity is achieved with every I/O access going through the VMM. Furthermore, the VMM can serve as an arbitrator/multiplexer/demultiplexer to implement useful features such as QoS control among VMs. However, VMM intervention also leads to longer I/O latency and higher CPU overhead due to the context switches between guest VMs and the VMM. Since the VMM serves as a central control point for all I/O accesses, it may also become a performance bottleneck for I/O intensive workloads.

Having device I/O access in the VMM also complicates the design of the VMM itself. It significantly limits the range of supported physical devices because new device drivers have to be developed to work within the VMM. To address this problem, VMware workstation [37] and Xen [13] carry out I/O operations in a host operating system or a special privileged VM called isolated device domain (IDD), which can run popular operating systems such as Windows and Linux that have a large number of existing device drivers. Although this approach can greatly simplify the VMM design and increase the range of supported hardware, it does not directly address performance issues with the approach used in VMware ESX Server. In fact, I/O accesses now may result in expensive operations called a world switch (a switch between the host OS and a guest VM) or a domain switch (a switch between two different VMs), which can lead to even worse I/O performance.

2.2 Overview of the Xen Virtual Machine Monitor

Xen is a popular high performance VMM. It uses para-virtualization [43], in which host operating systems need to be explicitly ported to the Xen architecture. This ar-

chitecture is similar to native hardware such as the x86 architecture, with only slight modifications to support efficient virtualization. Since Xen does not require changes to the application binary interface (ABI), existing user applications can run without any modification.

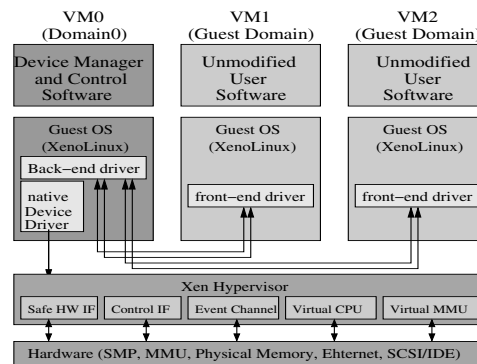


Figure 1: The structure of the Xen hypervisor, hosting three xenoLinux operating systems (courtesy [32])

Figure 1 illustrates the structure of a physical machine running Xen. The Xen hypervisor is at the lowest level and has direct access to the hardware. The hypervisor, instead of the guest operating systems, is running in the most privileged processor-level. Xen provides basic control interfaces needed to perform complex policy decisions. Above the hypervisor are the Xen domains (VMs). There can be many domains running simultaneously. Guest VMs are prevented from directly executing privileged processor instructions. A special domain called *domain0*, which is created at boot time, is allowed to access the control interface provided by the hypervisor. The guest OS in *domain0* hosts application-level management software and perform the tasks to create, terminate or migrate other domains through the control interface.

There is no guarantee that a domain will get a continuous stretch of physical memory to run a guest OS. Xen makes a distinction between *machine memory* and *pseudo-physical memory*. Machine memory refers to the physical memory installed in a machine, while pseudo-physical memory is a per-domain abstraction, allowing a guest OS to treat its memory as a contiguous range of physical pages. Xen maintains the mapping between the machine and the pseudo-physical memory. Only a certain parts of the operating system needs to understand the difference between these two abstractions. Guest OSes allocate and manage their own hardware page tables, with minimal involvement of the Xen hypervisor to ensure safety and isolation.

In Xen, domains can communicate with each other through shared pages and *event channels*. Event channels provide an asynchronous notification mechanism between domains. Each domain has a set of end-points

(or ports) which may be bounded to an event source. When a pair of end-points in two domains are bound together, a “send” operation on one side will cause an event to be received by the destination domain, which may in turn cause an interrupt. Event channels are only intended for sending notifications between domains. So if a domain wants to send data to another, the typical scheme is for a source domain to grant access to local memory pages to the destination domain. Then, these shared pages are used to transfer data.

Virtual machines in Xen usually do not have direct access to hardware. Since most existing device drivers assume they have complete control of the device, there cannot be multiple instantiations of such drivers in different domains for a single device. To ensure manageability and safe access, device virtualization in Xen follows a split device driver model [13]. Each device driver is expected to run in an *isolated device domain (IDD)*, which hosts a *backend* driver to serve access requests from guest domains. Each guest OS uses a *frontend* driver to communicate with the backend. The split driver organization provides security: misbehaving code in a guest domain will not result in failure of other guest domains. The split device driver model requires the development of frontend and backend drivers for each device class. A number of popular device classes such as virtual disk and virtual network are currently supported in guest domains.

2.3 OS-bypass I/O

Traditionally, device I/O accesses are carried out inside the OS kernel on behalf of application processes. However, this approach imposes several problems such as overhead caused by context switches between user processes and OS kernels and extra data copies which degrade I/O performance [5]. It can also result in *QoS crosstalk* [17] due to lacking of proper accounting for costs of I/O accesses carried out by the kernel on behalf of applications.

To address these problems, a concept called user-level communication was introduced by the research community. One of the notable features of user-level communication is *OS-bypass*, with which I/O (communication) operations can be achieved directly by user processes without involvement of OS kernels. OS-bypass was later adopted by commercial products, many of which have become popular in areas such as high performance computing where low latency is vital to applications. It should be noted that OS-bypass does not mean all I/O operations bypass the OS kernel. Usually, devices allow OS-bypass for frequent and time-critical operations while other operations, such as setup and management operations, can go through OS kernels and are handled by a privileged module, as illustrated in Figure 2.

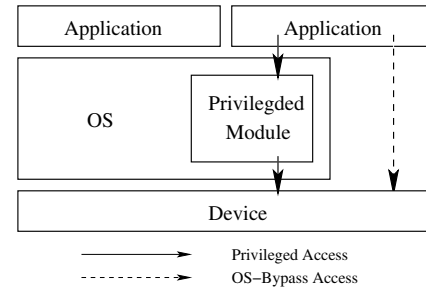


Figure 2: OS-Bypass Communication and I/O

The key challenge to implement OS-bypass I/O is to enable safe access to a device shared by many different applications. To achieve this, OS-bypass capable devices usually require more intelligence in the hardware than traditional I/O devices. Typically, an OS-bypass capable device is able to present virtual access points to different user applications. Hardware data structures for virtual access points can be encapsulated into different I/O pages. With the help of an OS kernel, the I/O pages can be mapped into the virtual address spaces of different user processes. Thus, different processes can access their own virtual access points safely, thanks to the protection provided by the virtual memory mechanism. Although the idea of user-level communication and OS-bypass was developed for traditional, non-virtualized systems, the intelligence and self-virtualizing characteristic of OS-bypass devices lend themselves nicely to a virtualized environment, as we will see later.

2.4 InfiniBand Architecture

InfiniBand [19] is a high speed interconnect offering high performance as well as features such as OS-bypass. InfiniBand host channel adapters (HCAs) are the equivalent of network interface cards (NICs) in traditional networks. InfiniBand uses a queue-based model for communication. A *Queue Pair (QP)* consists of a send queue and a receive queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data is to be placed. Communication instructions are described in *Work Queue Requests (WQR)*, or descriptors, and submitted to the queue pairs. The completion of the communication is reported through *Completion Queues (CQs)* using *Completion Queue Entries (CQEs)*. CQEs can be accessed by using polling or event handlers.

Initiating data transfers (posting descriptors) and notification of their completion (polling for completion) are time-critical tasks which use OS-bypass. In the Mellanox [21] approach, which represents a typical implementation of the InfiniBand specification, posting descriptors is done by ringing a doorbell. Doorbells are rung by writing to the registers that form the *User Ac-*

cess Region (UAR). Each UAR is a 4k I/O page mapped into a process's virtual address space. Posting a work request includes putting the descriptors to a QP buffer and writing the doorbell to the UAR, which is completed without the involvement of the operating system. CQ buffers, where the CQEs are located, can also be directly accessed from the process virtual address space. These OS-bypass features make it possible for InfiniBand to provide very low communication latency.

InfiniBand also provides a comprehensive management scheme. Management communication is achieved by sending management datagrams (MADs) to well-known QPs (QP0 and QP1).

InfiniBand requires all buffers involved in communication be registered before they can be used in data transfers. In Mellanox HCAs, the purpose of registration is two-fold. First, an HCA needs to keep an entry in the Translation and Protection Table (TPT) so that it can perform virtual-to-physical translation and protection checks during data transfer. Second, the memory buffer needs to be pinned in memory so that HCA can DMA directly into the target buffer. Upon the success of registration, a local key and a remote key are returned, which can be used later for local and remote (RDMA) accesses. QP and CQ buffers described above are just normal buffers that are directly allocated from the process virtual memory space and registered with HCA.

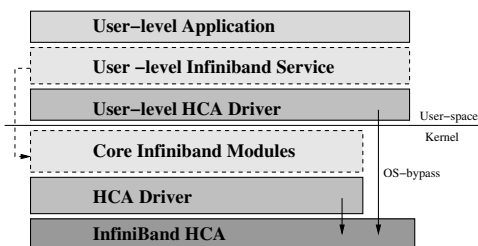


Figure 3: Architectural overview of OpenIB Gen2 stack

There are two popular stacks for InfiniBand drivers. VAPI [23] is the Mellanox implementation and OpenIB Gen2 [28] recently have come out as a new generation of IB stack provided by the OpenIB community. In this paper, our prototype implement is based on OpenIB Gen2, whose architecture is illustrated in Figure 3.

3 VMM-Bypass I/O

VMM-bypass I/O can be viewed as an extension to the idea of OS-bypass I/O in the context of VM environments. In this section, we describe the basic design of VMM-bypass I/O. Two key ideas in our design are *para-virtualization* and *high-level virtualization*.

In some VM environments, I/O devices are virtualized at the hardware level [37]. Each I/O instruction to access a device is virtualized by the VMM. With this ap-

proach, existing device drivers can be used in the guest VMs without any modification. However, it significantly increases the complexity of virtualizing devices. For example, one popular InfiniBand card (MT23108 from Mellanox [24]) presents itself as a PCI-X device to the system. After initialization, it can be accessed by the OS using memory mapped I/O. Virtualizing this device at the hardware level would require us to not only understand all the hardware commands issued through memory mapped I/O, but also implement a virtual PCI-X bus in the guest VM. Another problem with this approach is performance. Since existing physical devices are typically not designed to run in a virtualized environment, the interfaces presented at the hardware level may exhibit significant performance degradation when they are virtualized.

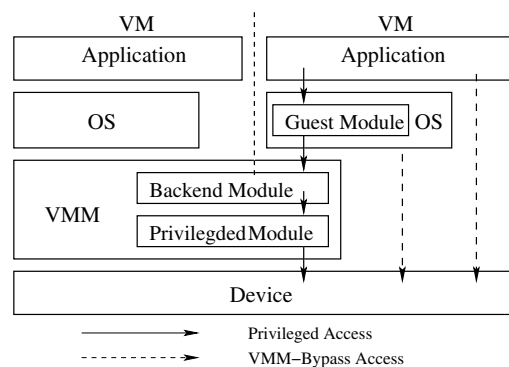


Figure 4: VM-Bypass I/O (I/O Handled by VMM Directly)

Our VMM-bypass I/O virtualization design is based on the idea of para-virtualization, similar to [11] and [44]. We do not preserve hardware interfaces of existing devices. To virtualize a device in a guest VM, we implement a device driver called *guest module* in the OS of the guest VM. The guest module is responsible for handling all the privileged accesses to the device. In order to achieve VMM-bypass device access, the guest module also needs to set things up properly so that I/O operations can be carried out directly in the guest VM. This means that the guest module must be able to create virtual access points on behalf of the guest OS and map them into the addresses of user processes. Since the guest module does not have direct access to the device hardware, we need to introduce another software component called *backend module*, which provides device hardware access for different guest modules. If devices are accessed inside the VMM, the backend module can be implemented as part of the VMM. It is possible to let the backend module talk to the device directly. However, we can greatly simplify its design by reusing the original privilege module of the OS-bypass device driver. In addi-

tion to serving as a proxy for device hardware access, the backend module also coordinates accesses among different VMs so that system integrity can be maintained. The VMM-bypass I/O design is illustrated in Figure 4.

If device accesses are provided by another VM (device driver VM), the backend module can be implemented within the device driver VM. The communication between guest modules and the backend module can be achieved through the inter-VM communication mechanism provided by the VM environment. This approach is shown in Figure 5.

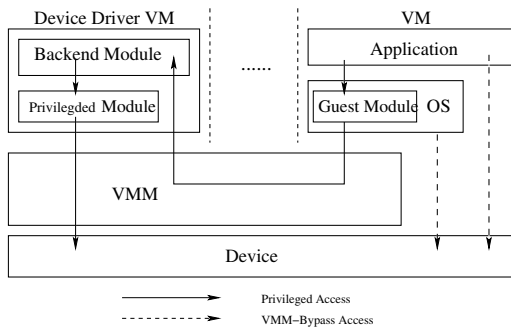


Figure 5: VM-Bypass I/O (I/O Handled by Another VM)

Para-virtualization can lead to compatibility problems because a para-virtualized device does not conform to any existing hardware interfaces. However, in our design, these problems can be addressed by maintaining existing interfaces which are at a higher level than the hardware interface (a technique we dubbed *high-level virtualization*). Modern interconnects such as InfiniBand have their own standardized access interfaces. For example, InfiniBand specification defines a *VERBS* interface for a host to talk to an InfiniBand device. The VERBS interface is usually implemented in the form of an API set through a combination of software and hardware. Our high-level virtualization approach maintains the same VERBS interface within a guest VM. Therefore, existing kernel drivers and applications that use InfiniBand will be able to run without any modification. Although in theory a driver or an application can bypass the VERBS interface and talk to InfiniBand devices directly, this seldom happens because it leads to poor portability due to the fact that different InfiniBand devices may have different hardware interfaces.

4 Prototype Design and Implementation

In this section, we present the design and implementation of Xen-IB, our InfiniBand virtualization driver for Xen. We describe details of the design and how we enable accessing the HCA from guest domains directly for time-critical tasks.

4.1 Overview

Like many other device drivers, InfiniBand drivers cannot have multiple instantiations for a single HCA. Thus, a split driver model approach is required to share a single HCA among multiple Xen domains.

Figure 6 illustrates a basic design of our Xen-IB driver. The backend runs as a kernel daemon on top of the native InfiniBand driver in the isolated device domain (IDD), which is domain0 is our current implementation. It waits for incoming requests from the frontend drivers in the guest domains. The frontend driver, which corresponds to the guest module mentioned in Section 3, replaces the kernel HCA driver in OpenIB Gen2 stack. Once the frontend is loaded, it establishes two event channels with the backend daemon. The first channel, together with shared memory pages, forms a device channel [13] which is used to process requests initiated from the guest domain. The second channel is used for sending InfiniBand CQ and QP events to the guest domain and will be discussed in detail later.

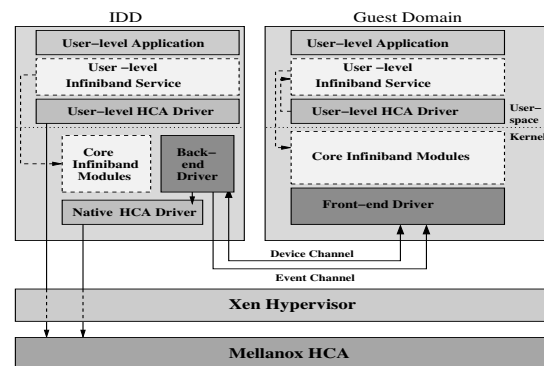


Figure 6: The Xen-IB driver structure with the split driver model

The Xen-IB frontend driver provides the same set of interfaces as a normal Gen2 stack for kernel modules. It is a relatively thin layer whose tasks include packing a request together with necessary parameters and sending it to the backend through the device channel. The backend driver reconstructs the commands, performs the operation using the native kernel HCA driver on behalf of the guest domain, and returns the result to the frontend driver.

The split device driver model in Xen poses difficulties for user-level direct HCA access in Xen guest domains. To enable VMM-bypass, we need to let guest domains have direct access to certain HCA resources such as the UARs and the QP/CQ buffers.

4.2 InfiniBand Privileged Accesses

In the following, we discuss in general how we support all privileged InfiniBand operations, including initialization, InfiniBand resource management, memory registra-

tion and event handling.

Initialization and resource management: Before applications can communicate using InfiniBand, it must finish several preparation steps including opening HCA, creating CQ, creating QP, and modifying QP status, etc. Those operations are usually not in the time critical path and can be implemented in a straightforward way. Basically, the guest domains forward these commands to the device driver domain (IDD) and wait for the acknowledgments after the operations are completed. All the resources are managed in the backend and the frontends refer to these resources by handles. Validation checks must be conducted in IDD to ensure that all references are legal.

Memory Registration: The InfiniBand specification requires all the memory regions involved in data transfers to be registered with the HCA. With Xen's paravirtualization approach, real machine addresses are directly visible to user domains. (Note that access control is still achieved because Xen makes sure a user domain cannot arbitrarily map a machine page.) Thus, a domain can easily figure out the DMA addresses of buffers and there is no extra need for address translation (assuming that no IOMMU is used). The information needed by memory registration is a list of DMA addresses that describes the physical locations of the buffers, access flags and the virtual address that the application will use when accessing the buffers. Again, the registration happens in the device domain. The frontend driver sends above information to the backend driver and get back the local and remote keys. Note that since the Translation and Protection Table (TPT) on HCA is indexed by keys, multiple guest domains are allowed to register with the same virtual address.

For security reasons, the backend driver can verify if the frontend driver offers valid DMA addresses belonging to the specific domain in which it is running. This check makes sure that all later communication activities of guest domains are within the valid address spaces.

Event Handling: InfiniBand supports several kinds of CQ and QP events. The most commonly used is the completion event. Event handlers are associated with CQs or QPs when they are created. An application can subscribe for event notification by writing a command to the UAR page. When those subscribed events happen, the HCA driver will first be notified by the HCA and then dispatch the event to different CQs or QPs according to the event type. Then the application/driver that owns the CQ/QP will get a callback on the event handler.

For Xen-IB, events are generated for the device domain, where all QPs and CQs are actually created. But the device domain cannot directly give a callback on the event handlers in the guest domains. To address this issue, we create a dedicated event channel between a fron-

tend and the backend driver. The backend driver associates a special event handler to each CQ/QP created due to requests from guest domains. Each time the HCA generates an event to these CQs/QPs, this special event handler gets executed and forwards information such as the event type and the CQ/QP identifier to the guest domain through the event channel. The frontend driver binds an event dispatcher as a callback handler to one end of the event channel after the channel is created. The event handlers given by the applications are associated to the CQs or QPs after they are successfully created. Frontend driver also maintains a translation table between the CQ/QP identifiers and the actual CQ/QPs. Once the event dispatcher gets an event notification from the backend driver, it checks the identifier and gives the corresponding CQ/QP a callback on the associated handler.

4.3 VMM-Bypass Accesses

In InfiniBand, QP accesses (posting descriptors) include writing WQEs to the QP buffers and ringing doorbells (writing to UAR pages) to notify the HCA. Then the HCA can use DMA to transfer the WQEs to internal HCA memory and perform the send/receive or RDMA operations. Once a work request is completed, HCA will put a completion entry (CQE) in the CQ buffer. In InfiniBand, QP access functions are used for initiating communication. To detect completion of communication, CQ polling can be used. QP access and CQ polling functions are typically used in the critical path of communication. Therefore, it is very important to optimize their performance by using VMM-bypass. The basic architecture of the VMM-bypass design is shown in Figure 7.

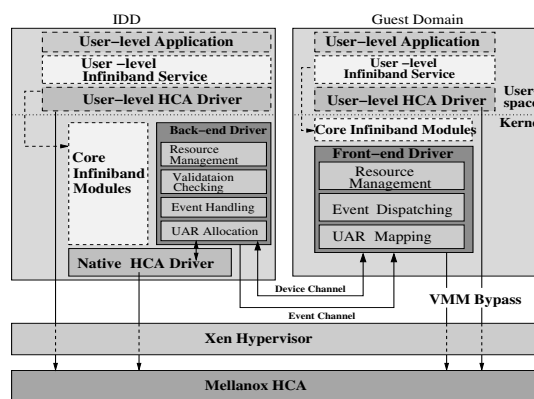


Figure 7: VMM-Bypass design of Xen-IB driver

Supporting VMM-bypass for QP access and CQ polling imposes two requirements on our design of Xen-IB: first, UAR pages must be accessible from a guest domain; second, both QP and CQ buffers should be directly visible in the guest domain.

When a frontend driver is loaded, the backend driver

allocates a UAR page and returns its page frame number (machine address) to the frontend. The frontend driver then remaps this page to its own address space so that it can directly access the UAR in the guest domain to serve requests from the kernel drivers. (We have applied a small patch to Xen to enable access to I/O pages in guest domains.) In the same way, when a user application starts, the frontend driver applies for a UAR page from the backend and remaps the page to the application's virtual memory address space, which can be later accessed directly from the user space. Since all UARs are managed in a centralized manner in the IDD, there will be no conflicts between UARs in different guest domains.

To make QP and CQ buffers accessible to guest domains, creating CQs/QPs has to go through two stages. In the first stage, QP or CQ buffers are allocated in the guest domains and registered through the IDD. During the second stage, the frontend sends the CQ/QP creation commands to the IDD along with the keys returned from the registration stage to complete the creation process. Address translations are indexed by keys, so in later operations the HCA can directly read WQRs from and write the CQEs back to the buffers (using DMA) located in the guest domains.

Since we also allocate UARs to user space applications in guest domains, the user level InfiniBand library now keeps its OS-bypass feature. The VMM-bypass IB-Xen workflow is illustrated in Figure 8.

It should be noted that since VMM-bypass accesses directly interact with the HCA, they are usually hardware dependent and the frontends need to know how to deal with different types of InfiniBand HCAs. However, existing InfiniBand drivers and user-level libraries already include code for direct access and it can be reused without spending new development efforts.

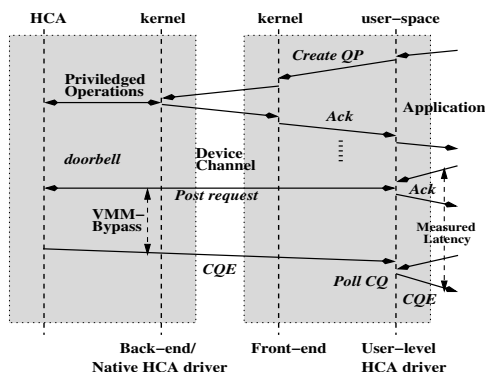


Figure 8: Working flow of the VMM-bypass Xen-IB driver

4.4 Virtualizing InfiniBand Management Operations

In an InfiniBand network, management and administrative tasks are achieved through the use of Management Datagrams (MADs). MADs are sent and received just like normal InfiniBand communication, except that they must use two well-known queue-pairs: QP0 and QP1. Since there is only one set of such queue pairs in every HCA, their access must be virtualized for accessing from many different VMs, which means we must treat them differently than normal queue-pairs. However, since queue-pair accesses can be done directly in guest VMs in our VMM-bypass approach, it would be very difficult to track each queue-pair access and take different actions based on whether it is a management queue-pair or a normal one.

To address this difficulty, we use the idea of high-level virtualization. This is based on the fact that although MAD is the basic mechanism for InfiniBand management, applications and kernel drivers seldom use it directly. Instead, different management tasks are achieved through more user-friendly and standard API sets which are implemented on top of MADs. For example, the kernel IPoIB protocol makes use of the subnet administration (SA) services, which are offered through a high-level, standardized SA API. Therefore, instead of tracking each queue-pair access, we virtualize management functions at the API level by providing our own implementation for guest VMs. Most functions can be implemented in a similar manner as privileged InfiniBand operations, which typically includes sending a request to the backend driver, executing the request (backend), and getting a reply. Since management functions are rarely in time-critical paths, the implementation will not bring any significant performance degradation. However, it does require us to implement every function provided by all the different management interfaces. Fortunately, there are only a couple of such interfaces and the implementation effort is not significant.

5 Discussions

In this section, we discuss issues related to our prototype implementation such as how safe device access is ensured, how performance isolation between different VMs can be achieved, and challenges in implementing VM check-pointing and migration with VMM-bypass. We also point out several limitations of our current prototype and how we can address them in future.

5.1 Safe Device Access

To ensure that accesses to virtual InfiniBand devices by different VMs will not compromise system integrity, we need to make sure that both privileged accesses and VMM-bypass accesses are safe. Since all privileged

accesses need to go through the backend module, access checks are implemented there to guarantee safety. VMM-bypass operations are achieved through accessing the memory-mapped UAR pages which contain virtual access points. Setting-up these mappings is privileged and can be checked. InfiniBand allows using both virtual and physical addresses for sending and receiving messages or carrying out RDMA operations, as long as a valid memory key is presented. Since the key is obtained through InfiniBand memory registration, which is also a privileged operation, we implement necessary safety checks in the backend module to ensure that a VM can only carry out valid memory registration operations. It should be noted that once a memory buffer is registered, its physical memory pages cannot be reclaimed by the VMM. Therefore, we should limit the total size of buffers that can be registered by a single VM. This limit check can also be implemented in the backend module.

Memory registration is an expensive operation in InfiniBand. In our virtual InfiniBand implementation, memory registration cost is even higher due to inter-domain communication. This may lead to performance degradation in cases where buffers cannot be registered in advance. Techniques such as *pin-down cache* can be applied when buffers are reused frequently, but it is not always effective. To address this issue, some existing InfiniBand kernel drivers create and use a *DMA key* through which all physical pages can be accessed. Currently, our prototype supports DMA keys. However, this leaves a security hole because all physical memory pages (including those belonging to other VMs) can be accessed. In future, we plan to address this problem by letting the DMA keys only authorize access to physical pages in the current VM. However, this also means that we need to update the keys whenever the VMM changes the physical pages allocated to a VM.

5.2 Performance Isolation

Although our current prototype does not yet implement performance isolation or QoS among different VMs, this issue can be addressed by taking advantage of QoS mechanisms which are present in the current hardware. For example, Mellanox InfiniBand HCAs support a QoS scheme in which a weighted round-robin algorithm is used to schedule different queue-pairs. In this scheme, QoS policy parameters are assigned when queue-pairs are created and initialized. After that, the HCA hardware is responsible for taking necessary steps to ensure QoS policies. Since queue-pair creations are privileged, we can create desired QoS policies in the backend when queue-pairs are created. These QoS policies will later be enforced by device hardware. We plan to explore more along this direction in future.

5.3 VM Check-pointing and Migration

VMM-bypass I/O poses new challenges for implementing VM check-pointing and migration. This is due to two reasons. First, the VMM does not have complete knowledge of VMs with respect to device accesses. This is in contrast to traditional device virtualization approaches in which the VMM is involved in every I/O operation and it can easily suspend and buffer these operations when check-pointing or migration starts. The second problem is that VMM-bypass I/O exploits intelligent devices which can store a large part of the VM system states. For example, an InfiniBand HCA has onboard memory which stores information such as registered buffers, queue-pair data structures, and so on. Some of the state information on an HCA can only be changed as side effects of VERBS functions calls. It does not allow changing it in an arbitrary way. This makes it difficult for check-pointing and migrations because when a VM is restored from a previous checkpoint or migrated to another node, the corresponding state information on the HCA needs to be restored also.

There are two directions to address the above problems. The first one is to involve VMs in the process of check-pointing and migration. For example, the VMs can bring themselves to some determined states which simplify check-pointing and migration. Another way is to introduce some hardware/firmware changes. We are currently working on both directions.

6 Performance Evaluation

In this section, we first evaluate the performance of our Xen-IB prototype using a set of InfiniBand layer micro-benchmarks. Then, we present performance results for the IPoIB protocol based on Xen-IB. We also provide performance numbers of MPI on Xen-IB at both micro-benchmark and application levels.

6.1 Experimental Setup

Our experimental testbed is an InfiniBand cluster. Each system in the cluster is equipped with dual Intel Xeon 3.0GHz CPUs, 2 GB memory and a Mellanox MT23108 PCI-X InfiniBand HCA. The PCI-X buses on the systems are 64 bit and run at 133 MHz. The systems are connected with an InfiniScale InfiniBand switch. The operating systems are RedHat AS4 with 2.6.12 kernel. Xen 3.0 is used for all our experiments, with each guest domain ran with single virtual CPU and 512 MB memory.

6.2 InfiniBand Latency and Bandwidth

In this subsection, we compared user-level latency and bandwidth performance between Xen-IB and native InfiniBand. Xen-IB results were obtained from two guest domains on two different physical machines. Polling was used for detecting completion of communication.

The latency tests were carried out in a ping-pong fashion. They were repeated many times and the average half round-trip time was reported as one-way latency. Figures 9 and 10 show the latency for InfiniBand RDMA write and send/receive operations, respectively. There is very little performance difference between Xen-IB and native InfiniBand. This is because in the tests, InfiniBand communication was carried out by directly accessing the HCA from the guest domains with VMM-bypass. The lowest latency achieved by both was around $4.2 \mu s$ for RDMA write and $6.6 \mu s$ for send/receive.

In the bandwidth tests, a sender sent a number of messages to a receiver and then waited for an acknowledgment. The bandwidth was obtained by dividing the number of bytes transferred from the sender by the elapsed time of the test. From Figures 11 and 12, we again see virtually no difference between Xen-IB and native InfiniBand. Both of them were able to achieve bandwidth up to 880 MByte/s, which was limited by the bandwidth of the PCI-X bus.

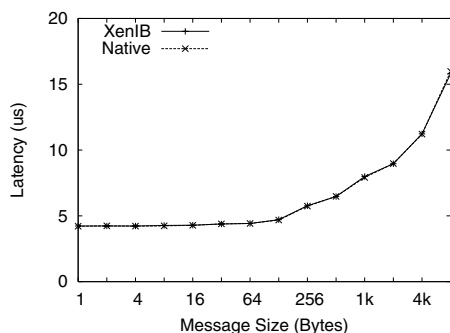


Figure 9: InfiniBand RDMA Write Latency

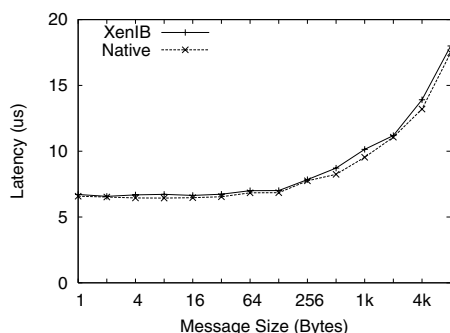


Figure 10: InfiniBand Send/Receive Latency

6.3 Event/Interrupt Handling Overhead

The latency numbers we showed in the previous subsection were based on polling schemes. In this section, we characterize the overhead of event/interrupt handling

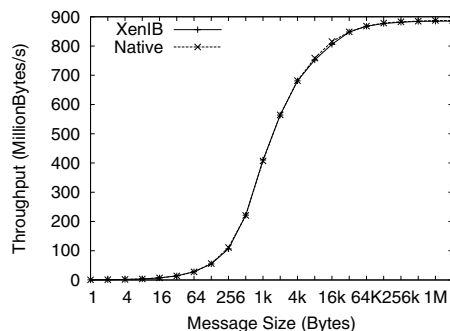


Figure 11: InfiniBand RDMA Write Bandwidth

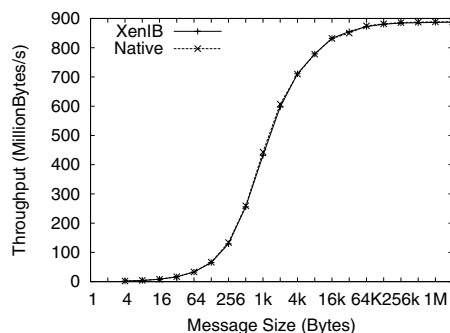


Figure 12: InfiniBand Send/Receive Bandwidth

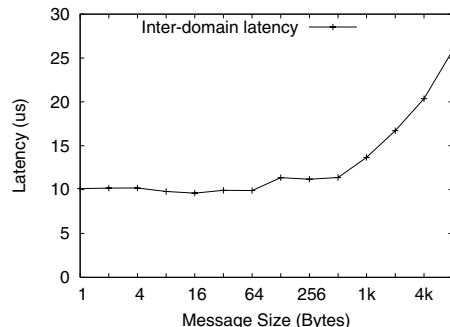


Figure 13: Inter-domain Communication One Way Latency

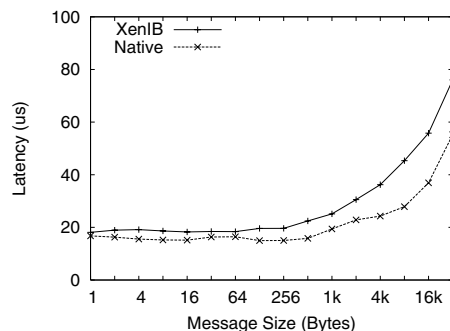


Figure 14: Send/Receive Latency Using Blocking VERBS Functions

in Xen-IB by showing send/receive latency results with blocking InfiniBand user-level VERBS functions.

Compared with native InfiniBand event/interrupt processing, Xen-IB introduces extra overhead because it requires forwarding an event from domain0 to a guest domain, which involves Xen inter-domain communication. In Figure 13, we show performance of Xen inter-domain communication. We can see that the overhead increases with the amount of data transferred. However, even with very small messages, there is an overhead of about $10\ \mu\text{s}$.

Figure 14 shows the send/receive one-way latency using blocking VERBS. The test is almost the same as the send/receive latency test using polling. The difference is that a process will block and wait for a completion event instead of busy polling on the completion queue. From the figure, we see that Xen-IB has higher latency due to overhead caused by inter-domain communication. For each message, Xen-IB needs to use inter-domain communication twice, one for send completion and one for receive completion. For large messages, we observe that the difference between Xen-IB and native InfiniBand is around $18\text{--}20\ \mu\text{s}$, which is roughly twice the inter-domain communication latency. However, for small messages, the difference is much less. For example, native InfiniBand latency is only $3\ \mu\text{s}$ better for 1 byte messages. This difference gradually increases with message sizes until it reaches around $20\ \mu\text{s}$. Our profiling reveals that this is due to “event batching”. For small messages, the inter-domain latency is much higher than InfiniBand latency. Thus, when a send completion event is delivered to a guest domain, a reply may have already come back from the other side. Therefore, the guest domain can process two completions with a single inter-domain communication operation, which results in reduced latency. For small messages, event batching happens very often. As message size increases, it becomes less and less frequent and the difference between Xen-IB and native IB increases.

6.4 Memory Registration

Memory registration is generally a costly operation in InfiniBand. Figure 15 shows the registration time of Xen-IB and native InfiniBand. The benchmark registers and unregisters a trunk of user buffers multiple times and measures the average time for each registration.

As we can see from the graph, Xen-IB adds consistently around 25%–35% overhead to the registration cost. The overhead increases with the number of pages involved in registration. This is because Xen-IB needs to use inter-domain communication to send a message which contains machine addresses of all the pages. The more pages we register, the bigger the size of message we need to send to the device domain through the inter-domain device channel. This observation indicates that

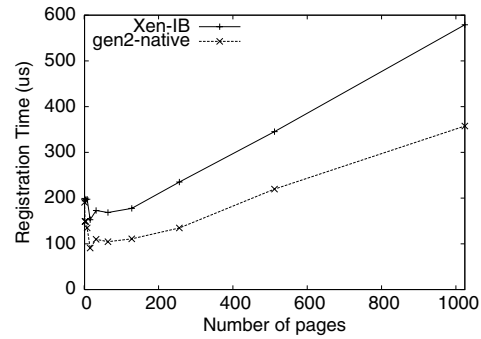


Figure 15: Memory Registration Time

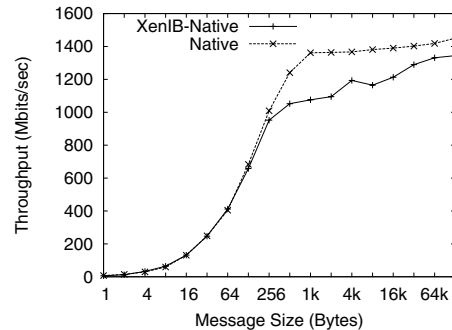


Figure 16: IPoIB Netperf Throughput

if the registration is a time critical operation of an application, we need to use techniques such as an efficient implementation of registration cache [38] to reduce costs.

6.5 IPoIB Performance

IPoIB allows one to run TCP/IP protocol suites over InfiniBand. In this subsection, we compared IPoIB performance between Xen-IB and native InfiniBand using Netperf [2]. For Xen-IB performance, the netperf server is hosted in a guest domain with Xen-IB while the client process is running with native InfiniBand.

Figure 16 illustrates the bulk data transfer rates over TCP stream using the following commands:

```
netperf -H $host -l 60 -- -s$size -S$size
```

Due to the increased cost of interrupt/event processing, we cannot achieve the same throughput while the server is hosted with Xen-IB compared with native InfiniBand. However, Xen-IB is still able to reach more than 90% of the native InfiniBand performance for large messages.

We notice that IPoIB achieved much less bandwidth compared with raw InfiniBand. This is because of two reasons. First, IPoIB uses InfiniBand unreliable datagram service, which has significantly lower bandwidth than the more frequently used reliable connection service due to the current implementation of Mellanox HCAs. Second, in IPoIB, due to the limit of MTU, large mes-

sages are divided into small packets, which can cause a large number of interrupts and degrade performance.

Figure 17 shows the request/response performance measured by Netperf (transactions/second) using:

```
netperf -l 60 -H $host -tTCP_RR -- -r $size,$size
```

Again, Xen-IB performs worse than native InfiniBand, especially for small messages where interrupt/event cost plays a dominant role for performance. Xen-IB performs more comparable to native InfiniBand for large messages.

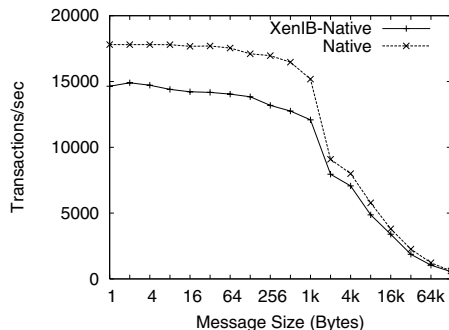


Figure 17: Netperf Transaction Test

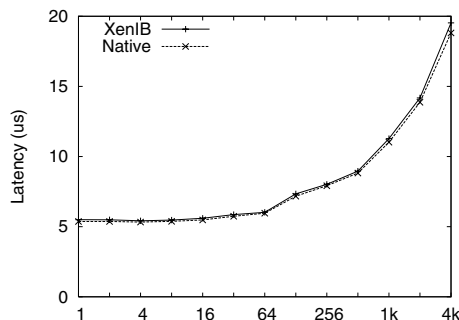


Figure 18: MPI Latency

6.6 MPI Performance

MPI is a communication protocol used in high performance computing. For tests in this subsection, we have used MVAPICH [27, 20], which is a popular MPI implementation over InfiniBand.

Figures 18 and 19 compare Xen-IB and native InfiniBand in terms of MPI one-way latency and bandwidth. The tests were run between two physical machines in the cluster. Since MVAPICH uses polling for all underlying InfiniBand communication, Xen-IB was able to achieve the same performance as native InfiniBand by using VMM-bypass. The smallest latency achieved by MPI with Xen-IB was 5.4 μ s. The peak bandwidth was 870 MBytes/s.

Figure 20 shows performance of IS, FT, SP and BT applications from the NAS Parallel Benchmarks suite [26]

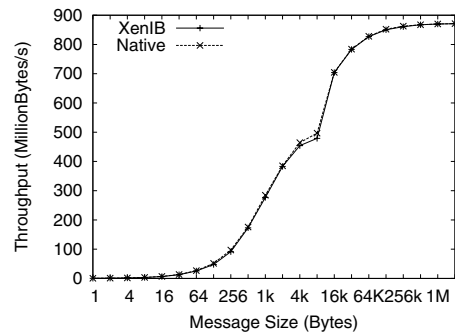


Figure 19: MPI Bandwidth

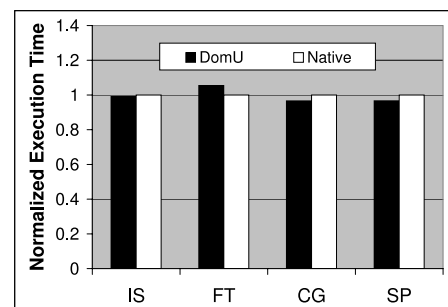


Figure 20: MPI NAS Benchmarks

(class A), which is frequently used by researchers in the area of high performance computing. We show normalized execution time based on native InfiniBand. In these tests, two physical nodes were used with two guest domains per node for Xen-IB. For native InfiniBand, two MPI processes were launched for each node. We can see that Xen-IB performs comparably with native InfiniBand, even for communication intensive applications such as IS. IB-Xen performs about 4% worse for FT and around 2–3% better for SP and BT. We believe the difference is due to the fact that MVAPICH uses shared memory communication for processes in a single node. Although MVAPICH with Xen-IB currently does not have this feature, it can be added by taking advantage of the page sharing mechanism provided by Xen.

7 Related Work

In Section 2.1, we have discussed current I/O device virtualization approaches such as those in VMware Workstation [37], VMware ESX Server [42], and Xen [13]. All of them require the involvement of the VMM or a privileged VM to handle every I/O operation. In our VMM-bypass approach, many time-critical I/O operations can be executed directly by guest VMs. Since this method makes use of intelligence in modern high speed network interfaces, it is limited to a relatively small range

of devices which are used mostly in high-end systems. The traditional approaches can be applied to a much wider ranges of devices.

OS-bypass is a feature found in user-level communication protocols such as active messages [41], U-Net [40], FM [29], VMMC [6], and Arsenic [33]. Later, it was adopted by the industry [12, 19] and found its way into commercial products [25, 34]. Our work extends the idea of OS-bypass to VM environments. With VMM-bypass, I/O and communication operations can be initiated directly by user space applications, bypassing the guest OS, the VMM, and the device driver VM. VMM-bypass also allows an OS in a guest VM to carry out many I/O operations directly, although virtualizing interrupts still needs the involvement of the VMM.

The idea of direct device access from a VM has been proposed earlier. For example, [7] describes a method to implement direct I/O access from a VM for IBM mainframes. However, it requires an I/O device to be dedicated to a specific VM. The VMM-bypass approach not only enables direct device access, but allows for safe device sharing among many different VMs. Recently, the industry has started working on standardization of I/O virtualization by extending the PCI Express standard [30] to allow a physical device to present itself as multiple virtual devices to the system [31]. This approach can potentially allow a VM to directly interact with a virtual device. However, it requires building new hardware support into PCI devices while our VMM-bypass approach is based on existing hardware. At about the same time when we were working on our virtualization support for InfiniBand in Xen, others in the InfiniBand community proposed similar ideas [39, 22]. However, details regarding their implementations are currently not available.

Our InfiniBand virtualization support for Xen uses a para-virtualization approach. As a technique to improve VM performance by introducing small changes in guest OSes, para-virtualization has been used in many VM environments [8, 16, 44, 11]. Essentially, para-virtualization presents a different abstraction to the guest OSes than native hardware, which lends itself to easier and faster virtualization. The same idea can be applied to the virtualization of both CPU and I/O devices. Para-virtualization usually trades compatibility for enhanced performance. However, our InfiniBand virtualization support achieves both high performance and good compatibility by maintaining the same interface as native InfiniBand drivers at a higher level than hardware. As a result, our implementation is able to support existing kernel drivers and user applications. Virtualization at higher levels than native hardware is used in a number of other systems. For example, novel operating systems such as Mach [15], K42 [4], and L4 [18] use OS level API or ABI emulation to support traditional OSes such as Unix

and Linux. Several popular VM projects also use this approach [10, 3].

8 Conclusions and Future Work

In this paper, we presented the idea of VMM-bypass, which allows time-critical I/O commands to be processed directly in guest VMs without involvement of a VMM or a privileged VM. VMM-bypass can significantly improve I/O performance in VMs by eliminating context switching overhead between a VM and the VMM or two different VMs caused by current I/O virtualization approaches. To demonstrate the idea of VMM-bypass, we described the design and implementation of Xen-IB, an VMM-bypass capable InfiniBand driver for the Xen VM environment. Xen-IB runs with current InfiniBand hardware and does not require modification to applications or kernel drivers which use InfiniBand. Our performance evaluations showed that Xen-IB can provide performance close to native hardware under most circumstances, with expected degradation on event/interrupt handling and memory registration.

Currently, we are working on providing check-pointing and migration support for our Xen-IB prototype. We are also investigating how to provide performance isolation by implementing QoS support in Xen-IB. In future, we plan to study the possibility to introduce VMs into high performance computing area. We will explore how to take advantages of Xen to provide better support of check-pointing, QoS and cluster management with minimum loss of computing power.

Acknowledgments

We would like to thank Charles Schulz, Orran Krieger, Muli Ben-Yehuda, Dan Poff, Mohammad Banikazemi, and Scott Guthridge of IBM Research for valuable discussions and their support for this project. We thank Ryan Harper and Nivedita Singhvi of IBM Linux Technology Center for extending help to improve the Xen-IB implementation. We also thank the anonymous reviewers for their insightful comments.

This research is supported in part by the following grants and equipment donations to the Ohio State University: Department of Energy's Grant #DE-FC02-01ER25506; National Science Foundation grants #CNS-0403342 and #CCR-0509452; grants from Intel, Mellanox, Sun, Cisco, and Linux Network; and equipment donations from Apple, AMD, IBM, Intel, Microway, Pathscale, Silverstorm and Sun.

References

- [1] IP over InfiniBand Working Group. <http://www.ietf.org/html.charters/ipoib-charter.html>.
- [2] Netperf. <http://www.netperf.org>.
- [3] D. Aloni. Cooperative Linux. <http://www.colinux.org>.

- [4] J. Appavoo, M. Auslander, M. Burtico, D. D. Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenberg, R. W. Wisniewski, and J. Xenidis. K42: an Open-Source Linux-Compatible Scalable Operating System Kernel. *IBM Systems Journal*, 44(2):427–440, 2005.
- [5] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, pages 53–60, November 1998.
- [6] M. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesalina. Virtual-Memory-Mapped Network Interfaces. In *IEEE Micro*, pages 21–28, Feb. 1995.
- [7] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk. Multiple Operating Systems on One Processor Complex. *IBM System Journal*, 28(1):104–123, 1989.
- [8] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.
- [9] P. M. Chen and B. D. Noble. When virtual is better than real. *Hot Topics in Operating Systems*, pages 133–138, 2001.
- [10] J. Dike. User Mode Linux. <http://user-mode-linux.sourceforge.net>.
- [11] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.
- [12] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–76, March/April 1998.
- [13] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *Proceedings of OASIS ASPLOS Workshop*, 2004.
- [14] R. P. Goldberg. Survey of Virtual Machine Research. *Computer*, pages 34–45, June 1974.
- [15] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid. UNIX as an application program. In *USENIX Summer*, pages 87–95, 1990.
- [16] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262, 2000.
- [17] S. M. Hand. Self-paging in the nemesis operating system. In *Operating Systems Design and Implementation, USENIX*, pages 73–86, 1999.
- [18] H. Hartig, M. Hohmuth, J. Liedtke, and S. Schonberg. The performance of micro-kernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 66–77, December 1997.
- [19] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2.
- [20] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Proceedings of 17th Annual ACM International Conference on Supercomputing (ICS '03)*, June 2003.
- [21] Mellanox Technologies. <http://www.mellanox.com>.
- [22] Mellanox Technologies. I/O Virtualization with InfiniBand. http://www.xensource.com/company/xensummit.html/Xen_Virtualization_InfiniBand_Mellanox_MKagan.pdf.
- [23] Mellanox Technologies. Mellanox IB-Verbs API (VAPI), Rev. 1.00.
- [24] Mellanox Technologies. Mellanox InfiniBand InfiniHost MT23108 Adapters. <http://www.mellanox.com>, July 2002.
- [25] Myricom, Inc. Myrinet. <http://www.myri.com>.
- [26] NASA. NAS Parallel Benchmarks. <http://www.nas.nasa.gov/Software/NPB/>.
- [27] Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand on VAPI Layer. <http://nowlab.cse.ohio-state.edu/projects/mpi-iba/index.html>.
- [28] Open InfiniBand Alliance. <http://www.openib.org>.
- [29] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.
- [30] PCI-SIG. PCI Express Architecture. <http://www.pcisig.com>.
- [31] PCI-SIG. PCI I/O Virtualization. http://www.pcisig.com/news_room/news/press_releases/2005_06_06.
- [32] I. Pratt. Xen Virtualization. Linux World 2005 Virtualization BOF Presentation.
- [33] I. Pratt and K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *INFOCOM*, pages 67–76, 2001.
- [34] Quadrics, Ltd. QsNet. <http://www.quadrics.com>.
- [35] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, pages 39–47, May 2005.
- [36] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI—The Complete Reference. Volume 1 - The MPI-1 Core, 2nd edition*. The MIT Press, 1998.
- [37] J. Sugerman, G. Venkitachalam, and B. H. Lim. Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor. In *Proceedings of USENIX*, 2001.
- [38] H. Tezuka, F. O’Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Proceedings of the 12th International Parallel Processing Symposium*, 1998.
- [39] Voltaire. Fast I/O for Xen using RDMA Technologies. http://www.xensource.com/company/xensummit.html/Xen_RDMA_Voltaire_YHaviv.pdf.
- [40] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.
- [41] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, 1992.
- [42] C. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.
- [43] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference, Monterey, CA*, June 2002.
- [44] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of 5th USENIX OSDI, Boston, MA*, Dec 2002.

Provenance-Aware Storage Systems

Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, Margo Seltzer
Harvard University

Abstract

A Provenance-Aware Storage System (PASS) is a storage system that automatically collects and maintains *provenance* or *lineage*, the complete history or ancestry of an item. We discuss the advantages of treating provenance as meta-data collected and maintained by the storage system, rather than as manual annotations stored in a separately administered database. We describe a PASS implementation, discussing the challenges it presents, performance cost it incurs, and the new functionality it enables. We show that with reasonable overhead, we can provide useful functionality not available in today's file systems or provenance management systems.

1 Introduction

Provenance is traditionally the ownership history of an object. In digital systems, ownership history includes a description of how the object was derived [4]. In software development, build rules express provenance and source code control systems track it. Archivists maintain provenance meta-data to support document viability, renderability, understandability, authenticity, and identity in preservation contexts [21]. Scientific reproducibility requires provenance to identify precisely input data sets, experimental procedures, and parameterization. The business community uses “lineage” to refer to the history of a document. In all of these domains provenance increases an object's value.

Digital provenance is typically stored in standalone database systems, maintained in parallel with the data to which it refers [2, 6, 25]. Separating provenance from its data introduces problems such as: ensuring consistency between the provenance and the data, enforcing provenance maintenance, and preserving provenance during backup, restoration, copies, etc.

Provenance should be maintained by the storage system, since provenance is merely meta-data and storage

systems manage meta-data. Managing provenance in the storage system provides many advantages:

- The storage system can generate system-level provenance automatically, freeing users from manually tracking provenance and tool designers from constructing provenance-aware applications.
- Provenance collection and management are transparent. Users take no special actions, so provenance collection is the norm rather than the exception.
- The tight coupling between data and provenance provided by the storage system avoids provenance loss both during normal activity and during management procedures such as backup, restoration, or data migration.
- System-level provenance produces a level of meta-data completeness difficult or impossible to achieve with application-level solutions.

A *provenance-aware storage system* (PASS) is a storage system that automatically collects, stores, manages, and provides search for provenance. Provenance-aware storage offers functionality unavailable in conventional systems. For example, comparing the provenance of two pieces of derived data, such as simulation results, can reveal changes between two program invocations. We can use provenance to identify the particular workflow that produced a document; this provides a tight coupling between workflow management and information life-cycle management (ILM). Searchable provenance enables queries like “Who is using my dataset?” or “On whose data am I depending when I run this experiment?” System-level provenance enables identification of configuration changes that affect applications.

We present the design and evaluation of a prototype PASS. A PASS is both a provenance solution and substrate upon which we can support other provenance systems. Domains characterized by command-line invoca-

tion of data transformations are well served by PASS and may not require any domain-specific solution. Domains that require GUI-driven applications or application environments still derive benefit from integration with provenance-aware storage. This paper contributes: An approach to automatic provenance collection and maintenance; a prototype that demonstrates the efficacy and practicality of the PASS concept; an evaluation of PASS functionality and overhead; and a research agenda for future work on PASS.

The rest of this paper is organized as follows. In Section 2, we present several novel use cases that PASS enables. In Section 3, we compare PASS to existing provenance systems. In Section 4, we outline the requirements for PASS and explain how PASS interacts with existing provenance systems and file system utilities. In Section 5, we describe our PASS implementation. In Section 6, we quantify the overheads PASS introduces. We conclude in Section 7 with a discussion of open research issues and the long-term PASS agenda.

2 Novel Functionality

PASS provides novel features not available in other systems. In this section, we present use cases and examples that demonstrate the power of provenance-aware storage.

2.1 Script Generation

Early PASS users were most excited by PASS's potential to create scripts that reproduce a file originally created through trial and error. We find this one of the most compelling features of PASS.

Recently one of the authors was preparing a paper and discovered that a PostScript figure was being generated with the wrong bounding box, producing a tiny figure surrounded by blank space. After several hours of experimental tinkering as the submission deadline approached, he discovered that creating a huge bitmap, cropping it with an image-manipulation tool, and then converting it to a different format for incorporation into the paper solved the problem. Discovering the precise solution was an iterative multistage process. When the figure finally displayed properly, its provenance allowed automating the processing. Currently, we generate scripts to reproduce a particular object; generalizing these scripts so that they transform objects of one type into objects of another type is a future project.

2.2 Detecting System Changes

Because we collect provenance at the system level, we know the operating system, library versions, and the environment present when an object was created. For ex-

ample, the output of the `sort` utility depends on the value of the environment variable `LC_COLLATE`. Examining the provenance of files produced by `sort` explains why two seemingly identical invocations of the same command on the same input produce different results. Because we record the process' environment in the provenance, it becomes easy to identify what differs between the two invocations. Tracking changes in libraries and tools is similarly straightforward. The provenance that PASS collects reveals changes in the environment, libraries, operating system, or tools.

2.3 Intrusion Detection

Since a PASS collects provenance inside the operating system and file system, it provides a detailed record of how objects change. This feature can be used in some cases both for intrusion detection and subsequent forensic analysis.

One of the authors uses a UNIX version of the Windows trash bin, aliasing `rm` (remove) to

```
mv !* ~/etc/garbage
```

which moves the target files to a garbage directory. Quite unexpectedly, several recently deleted files appeared in `~/public.html`. This happened on a non-PASS system, so it took nearly a half hour to discover that `~/etc/garbage` had been symbolically linked to `~/public.html`. Had this occurred on a PASS, the provenance of the files in `~/public.html` would have revealed that they had been explicitly moved from their original locations. Had the shell provided application-specific provenance, the `rm` alias would have been readily apparent.

2.4 Retrieving Compile-Time Flags

A common approach to understanding software performance is to recompile with different functionality selected via macros defined on the compiler command line. However, in order to interpret the results, one must keep track of which macros were defined for each program run. Often, users neglect to record this information; then, once the details have been forgotten, it becomes necessary to redo the work. PASS automatically records command-line arguments, as well as the relationship between the various versions of the program and the performance results. It is thus possible to recover the macro arguments and functionality choices.

2.5 Build Debugging

Large build environments are often missing elements of their dependency lists. These become apparent when a

build repeatedly produces a flawed executable. The traditional solution (suggested explicitly for the Linux kernel) is to make `clean` after any change.

PASS easily identifies an inconsistent build: in nearly all build environments, it is incorrect for two different versions of the same source or intermediate file to simultaneously be ancestors of the same output. Examining the complete ancestry of an obsolete file reveals immediately the intermediate file that should have been rebuilt, identifying the missing dependency.

2.6 Understanding System Dependencies

Early experimentation with PASS presented some surprising results. We found that some objects we created unexpectedly included `/bin/mount` in their provenance. This was baffling, since we were executing relatively simple commands such as `sort a > b`.

We discovered that the Linux C library frequently reads the mount table, `/etc/mtab`, to determine where `procfs` is mounted before using it. `mount` and `umount` update `/etc/mtab`, so these programs quite correctly appeared in our sorted file's ancestry. These files also appear in the provenance of *any* process that reads the mount table to find `procfs`.

This behavior is either good or bad, depending on perspective. For a system designer, such information is useful; to a user, it is distracting. To preserve the information for some without burdening others, we provide a provenance truncation utility, `ptrunc`, that removes `/etc/mtab`'s (or any file's) ancestry.

3 Provenance Solutions

Provenance is pervasive in scientific computing, business, and archival. Simmhan et al. categorize provenance solutions in terms of their architecture: database-oriented, service-oriented, and "other" [29]. We borrow and enrich this taxonomy. We extend database-oriented approaches to include file and file-system-oriented approaches such as PASS. The service-oriented architecture encompasses myriad grid-based solutions. Simmhan's "other" category refers to scripting architectures; we treat software development tools, i.e., source code control and build systems, as a specific instance of a scripting architecture. To these three categories, we add "environment architectures" where users perform all tasks in a unified environment that tracks provenance.

3.1 File, File System and Database Approaches

One obvious approach to provenance maintenance is to include provenance inside the corresponding data

file. Astronomy's Flexible Image Transport (FITS) format [18] and the Spatial Data Transfer Standard (SDTS) [23] are examples of this approach. A FITS file header consists of a collection of tagged attribute/value pairs, some of which are provenance. Whenever a file is transformed, additional provenance is added to this header. This approach addresses the challenge of making the provenance and data inseparable, but it introduces other disadvantages. It is expensive to search the attribute space to find objects meeting some criteria. Tools that operate on such files must read and write the headers and be provenance-aware. The validity and completeness of the provenance is entirely dependent upon the tools that process the data. Worse yet, there is no way to determine if provenance is complete or accurate.

The Lineage File System (LinFS) [15] is most similar to PASS. LinFS is a file system that automatically tracks provenance at the file system level, focusing on executables, command lines and input files as the only source of provenance, ignoring the hardware and software environment in which such processes run. As shown in Section 2, complete system-level provenance provides functionality unavailable in other systems. A second, and perhaps more important, difference is that LinFS delays provenance collection, performing it at user-level by writing it to an external database. In contrast, PASS manages its provenance database directly in the kernel, providing greater synchronicity between data and provenance; PASTA, our provenance-aware file system, manages both the data and provenance producing a tighter coupling than provided by a separate user-level database.

Trio [27] is to databases what a PASS is to file systems. Trio is a database system that incorporates uncertainty, managing both data and its provenance. It extends SQL to support lineage and accuracy information when requested by a user or application. Trio and PASS are complementary. Trio focuses on the formalism to describe uncertainty via lineage and operates on tuples within a database framework; PASS focuses on a less structured environment and operates on files.

3.2 Service-oriented Architectures

Many of the computational sciences use provenance systems designed for grid environments since provenance facilitates scientific verification, reproducibility, and collaboration. Most of these systems use a directed-acyclic-graph (DAG) representation to describe workflows. The tools that understand these workflows collect provenance and transmit it to a grid provenance service. For example, Globus [7] is used widely by high-energy physicists and includes the Metadata Catalog Service (MCS) [6] that stores metadata for logical data objects. MCS includes a set of common attributes and permits inclusion

of domain- or application-specific attributes. San Diego SuperComputer's Storage Request Broker [2, 26] has a Metadata Catalog similar to the MCS.

Chimera [8] is a virtual data system providing a virtual data language (VDL) and a virtual data catalog (VDC). The VDC implements a virtual data schema defining the objects and relations used to capture descriptions of program invocations and to record these invocations. The VDL is used for defining and manipulating data derivation procedures stored in the VDC. Chimera can be used to generate Grid workflows from the derivations.

These systems place the responsibility for provenance maintenance with the grid tools, storing provenance in a system parallel to the data storage. They create and maintain provenance only for data that is processed by provenance-aware tools, so there is no mechanism to capture provenance for local experimentation or operations issued outside the bounds of these tools. PASS provides capabilities not found in these systems and is complementary to their approaches. A grid-accessible PASS provides the advantages of both worlds.

3.3 Scripting Architectures

Software developers manage provenance manually using source code control and build systems. Though powerful, these systems rely more on manual intervention than PASS does. Build systems maintain dependencies *after* those dependencies have been specified; PASS derives dependencies based upon program execution. Source code control systems track differences between manually-declared versions, but a manually-entered commit message is typically the only conceptual expression of the transformation between those two versions. Thus, the quality of provenance in these systems depends on the quality of commit messages and build configuration files. For example, makefiles that track include dependencies properly are considerably more useful than those that do not.

The source code control systems most similar to PASS are ClearCase (and its predecessor DSEE) and Vesta. ClearCase [5] is a source code control system, and like PASS, it is based on a custom file system. The file system serves as the source code repository, and the build system relies on the standard `make` utility. The custom file system tracks and maintains system dependencies to avoid work in future builds and to trigger rebuilds. PASS also captures these dependencies. As is the case with all build systems of which we are aware, ClearCase requires that critical dependencies be specified *a priori*; PASS derives dependencies by observation.

Vesta [12] is a second generation build system developed at DEC Systems Research Center (SRC). The key design goals were making builds repeatable, consistent,

and incremental. As with DSEE, Vesta relies on a custom build environment that monitors the build process to extract dependencies and record complete environment information to facilitate repeatable builds. Like DSEE and other source code control systems, it relies on an *a priori* description of the derivation process. As a result, while extraordinarily useful for software development, it ignores the central PASS challenge: automatically generating the derivation rules as a system runs.

3.4 Environment Architectures

Other domains have environments that track work and record provenance as Vesta does for software development. GenePattern [10] is an environment for computational biologists, the Collaboratory for Multi-scale Chemical Sciences (CMCS) [20] is an environment for chemists, and the Earth System Science Workbench (ESSW) [9] is an environment for earth scientists. As long as a user modifies data exclusively in one of these environments, the environment can effectively track provenance. However, operating on data outside the environment or moving data between two different environments breaks the provenance chain. Traditional file system utilities, such as backup and restore, or regular utilities, such as remove and rename, can also break the provenance chain. The semantic information these environments provide is powerful; we propose PASS as a substrate under such environments. This hybrid architecture avoids disruptions in provenance and adds the ability to augment an environment's provenance with provenance about the operating system, libraries, and other system-level information.

3.5 Summary

Provenance-aware storage provides functionality not found in today's provenance solutions while compatible with most of them. Ultimately, end-to-end provenance requires multiple approaches working in concert. As shown by our evaluation, using PASS as a substrate for the end-to-end solution provides significant benefits.

4 The PASS Vision

PASS collects and maintains provenance for all the objects stored in it. We define provenance to be a description of the execution history that produced a persistent object (file). We represent provenance as a collection of attribute/value pairs, referenced by a unique identifier called a *pnode number*. Provenance can contain references to objects that are themselves *provenanced*, i.e., have provenance. Therefore, complete provenance is the transitive closure over all such references.

In both service-oriented architectures and PASS, provenance is captured in a DAG. PASS automatically generates the DAG describing the relationship between processes running on it and the files stored in it. Our prototype tracks provenance at a file granularity, but this is not an inherent property of PASS; a system could track provenance at finer or coarser granularities [16] (e.g., bytes, lines, directories or whole volumes).

Data on a PASS is considered to be either new data or the output of some process. The provenance of a process' output must include:

- A unique reference to the particular instance of the executable that created it.
- References to all input files.
- A complete description of the hardware platform on which the output was produced.
- A complete description of the operating system and system libraries that produced the output.
- The command line.
- The process environment.
- Parameters to the process (frequently encapsulated in the command line or input files)
- Other data (e.g., the random number generator seed) necessary to make pseudo-random computation repeatable.

Collecting all the information listed above poses challenges, but is possible. However, it is not possible to automatically collect provenance that the system never sees; we call such provenance *opaque provenance*. Opaque provenance arises when data originates from a non-PASS source, such as a user, another computer, or another file system that is not provenance-aware.

There are three approaches to opaque data. First, a PASS records any provenance it can deduce about an input, such as its creator, create time, its source (expressed as the global name for a networked object or the full path for a local, non-PASS object) and a unique fingerprint of the source (e.g., a hash value). Second, a PASS permits users to add annotations to files. Annotations provide either provenance for opaque data or semantic information, invisible to the system. Third, a PASS allows storage and retrieval of application-specific provenance. Application-specific provenance records are like annotations, but are provided programmatically from, for example, a driver for a remote data-producing device (e.g., a driver taking data from a telescope) or an application environment such as GenePattern [10]. A PASS distinguishes between internally-collected provenance, annotations, and application-generated provenance so queries can specify which attribute types to consider.

Field	Value
FILE	/b
ARGV	sort a
NAME	/bin/sort,/bin/cat
INPUT	(<i>pnode number of a</i>)
OPENNAME	/lib/i686/libc.so.6, /usr/share/locale/locale.alias, /etc/mtab,/proc/meminfo .#prov.mtab
ENV	PWD=/pass USER=root ...
KERNEL	Linux 2.4.29+autoprov #17 ...
MODULE	pasta,kbdb,autofs4,3c59x,...

Table 1: Provenance of `sort a > b`. The FILE record is relative to the root of the PASS volume.

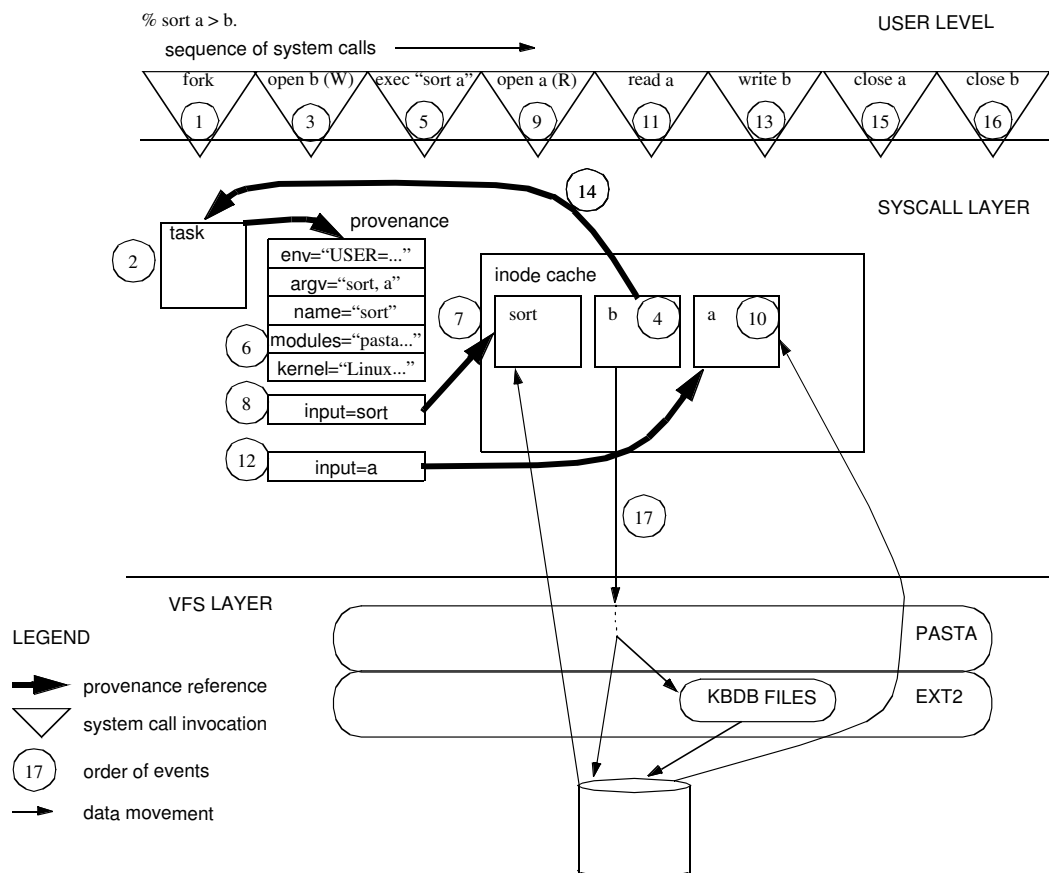
4.1 Requirements

PASS must automatically collect provenance and also provide the functionality outlined in this section.

PASS must support **application-generated provenance**, so that PASS can be used as a substrate for domain-specific provenance solutions. Applications that currently write to domain-specific provenance systems can instead write into a PASS's provenance database, producing an integrated view of application and system provenance. Our implementation uses a simple, low-level data representation that is easily mapped to XML, a relational schema, or any other data format used by an existing provenance solution.

PASS must provide **security for provenance**. Provenance requires access controls different from the data it describes. Consider an employee performance review that includes input from others: while the review itself must be readable by the employee, the provenance must not [15]. We conducted user studies to gather requirements for a provenance security model and determined that the security model for provenance can be divided into two pieces: one that protects access to ancestors and descendants and one that protects access to attributes. We are designing a security model for provenance, but its details are beyond the scope of this paper [3].

Finally, PASS must provide support for **queries on provenance**. Collecting provenance on data is not useful unless that provenance is easily accessed. Provenance queries fall into two categories: conventional attribute lookup and computation of transitive closures of ancestry (or descendancy) information. The latter are particularly challenging for most data management systems.



1. Create a new process via `fork`.
2. Materialize a `task_struct` structure for the new process.
3. Open output file `b` as stdout.
4. Place `b`'s inode into the inode cache.
5. Exec `sort`
6. Create provenance records for the process.
7. Load the executable; place its inode in the inode cache.
8. Associate the executable's inode to the running process.
9. Open input file `a`.
10. Read `a`'s inode into the inode cache.
11. Read from `a`.
12. Add file `a`'s provenance to the current process.
13. Write output to `b`.
14. Transfer process' provenance to `b`.
15. Close `a`.
16. Close `b`.
17. Write provenance to database.

Figure 1: Provenance Collection Example: We show the relevant system calls and provenance collection and maintenance steps for a simple command.

5 Implementation

We implemented PASS in Linux 2.4.29. To provide a framework for the implementation discussion, we begin with an example describing how PASS collects and maintains provenance. We then present an overview of provenance collection followed by the details of its implementation. We conclude this section by evaluating our prototype relative to the requirements presented in the previous section.

Throughout the rest of this section, we use the example command line “`sort a > b`” to describe how PASS collects and stores provenance. Table 1 shows the records that are added to the database, and Figure 1 shows the sequence of important system calls and the provenance collection steps performed while the command executes.

With this structure in mind, we present the details of our implementation in four parts. First, we present an overview of provenance collection. Second, we discuss the collector, which implements the activity shown in the the syscall layer in the figure. Third, we discuss our provenance-aware file system, which resides below the VFS layer. Last, we present our query system, which is not shown in the figure.

5.1 Overview

The system maintains provenance information both in memory and on disk, but the two representations do not map one-to-one. On disk, file ancestry can be expressed as cross-references to other files. In memory, however, PASS must account for processes and for other objects such as pipes or sockets that play important roles in provenance collection and derivation, but are not materialized in the file system.

Processes are provenanced objects, because processes produce files that must incorporate their creating process’ provenance. For example, in Figure 1, the process’ provenance is the collection of records that are attached to the task structure. Since our implementation does not track explicit data flow within a process, *all* data a process accesses can potentially affect the process’ outputs and must be included in its provenance.

In memory, the system maintains an ancestry graph in which files reference processes, and processes reference PASS files, pipes, non-PASS files, and (transitively) even other processes. Many of these objects are never materialized on disk, and some that are written to disk cannot be readily cross-referenced. When provenance is written to disk, in-memory references to persistent objects are recorded in the database as references while the provenance of a non-persistent referenced object (e.g., a pipe) is copied directly into the referring object’s provenance.

Therefore, each object on disk may correspond to several in-memory objects, and in-memory non-persistent objects may have their provenance recorded multiple times on disk if they are ancestors of multiple files.

The system collects provenance for every process, because it cannot know in advance which processes might ultimately write to a PASS volume. It also collects provenance for non-PASS files, which is retained as long as their inodes are kept in memory.

5.2 The Collector

One of the key challenges in PASS is translating a sequence of low-level events into a sequence of meaningful provenance entries; the collector performs this translation. It intercepts system calls, translating them into in-memory provenance records, which are then attached to key kernel data structures. Figure 1 shows how the collector translates some system calls into provenance records. For example, the `exec` system call (step 5) triggers creation of the ENV, ARGV, NAME, KERNEL, and MODULES provenance records in step 6. It also maintains the ancestry graph for in-memory objects. The final job of the collector is to map the in-memory graph to the on-disk provenance that is passed to the storage layer, shown as step 17 in Figure 1. We refer to on-disk provenance of an object via a unique *pnode number*.

In the `sort` example, the collector’s work is relatively simple. The collector generates a provenance record for each provenance-related system call and binds the record to the appropriate structure. Table 2 summarizes the system calls that the collector intercepts, the provenance records generated, and the structures to which records are attached.

Duplicate Elimination In the example, `sort` might issue many `read` system calls if its input file is large. The simple-minded collector creates a new INPUT provenance record for each read, even though the subsequent provenance records are exact duplicates of the first. Our collector tests for and eliminates duplicates at two points. It drops a new record when that record exactly duplicates an existing record already attached to the same kernel object. This does not eliminate all duplicates, however. Consider the shell command (`sort a; sort a`) `> b`, which writes two sorted copies of `a` into `b`. Both `sort` processes read `a` and have `a` as an ancestor. When the collector traverses the in-memory graph to write provenance for `b`, two references to `a` appear, so the collector also suppresses duplicates during this traversal. This duplicate elimination removes identical ancestors (e.g., the two occurrences of `a`), but since the two `sort` processes have different process ids, the

System Call	Record Type	Description	Where attached
execve	ENV	environment	current process
	ARGV	command line arguments	current process
	NAME	process name	current process
	PID	process id	current process
	KERNEL	kernel version	current process
	MODULES	kernel modules loaded	current process
	INPUT	reference to program	current process
open	OPENNAME	pathname of file (non-PASS files only)	target file's inode
read	INPUT	reference to file	current process
write	INPUT	reference to current process	target file's inode

Table 2: Basic provenance collection by system call.

provenance still reveals that there were two invocations of `sort`.

Versions Suppose that we executed `sort a > b` but `b` already existed. The system would truncate `b` and write new data to it, logically creating a new version of `b`. The system needs to ensure that there is a new version for the provenance of `b`, as the new `b` may have no relationship whatsoever to the old. Creating a new provenance version allocates a new pnode number. The collector intercepts the `truncate` operation to make sure this happens at the right time.

Because we are not running on a versioning file system, the data in the old `b` is lost. However, our file system does retain old versions of provenance, because the old `b` could be an ancestor of an important file, and we cannot remove that file's provenance.

In this example, it was easy to determine when to create new versions. However, declaring a new version is more complicated in the general case. Consider a sequence of write system calls. Does each write create a new version? What if a process closes and re-opens the file after each write?

The simple approach of declaring a new version for every write suffers from *provenance explosion*, the creation of too much provenance and it does not suggest when to create new versions for processes. The collector must be able to group multiple writes together into a single version. In the simple case, it is sufficient to declare a new version on a file's last `close` and on any `sync`. The next section discusses the solution for more complicated cases.

Cycles Ideally, only meaningful versions should be given new pnode numbers. Versions generated by the system as a result of the way the application is written or as a result of properties of the collector implemen-

tation should be minimized. However, as the workload increases in complexity, reducing the number of versions introduces an even more daunting challenge: cycles. Provenance cycles are problematic, because they represent violations of causality (i.e., an object depending on the existence of its children). Consider the program that does:

```
read a
write a
```

This program reads `a`, becoming a descendant of `a`. Then it writes `a`, making itself also an ancestor of `a`. This produces a cycle unless the write to `a` creates a new version that is not an ancestor of the process.

This particular case is easily suppressed; however, the problem is considerably broader than this. Consider two processes `P` and `Q`:

<code>P</code>	<code>Q</code>
<code>read a</code>	
	<code>read b</code>
<code>write b</code>	
	<code>write a</code>
<code>close a</code>	<code>close a</code>
<code>close b</code>	<code>close b</code>

If no new versions are created until `close`, `Q`'s write of `a` creates a cycle; `Q` is an ancestor of `a`, which is an ancestor of `P`, which is an ancestor of `b`, which is an ancestor of `Q`. More complex examples include dozens of processes and intervening files.

Since we want to avoid creating a new version on every write, there are at least three possible approaches:

1. Ignore it. Let cycles appear in the database and make the query tools deal with them. We consider this approach unacceptable.
2. Declare a new version only on writes that add new provenance information. This approach generates a

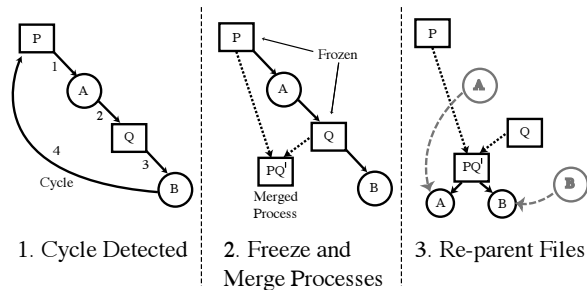


Figure 2: Cycle breaking by node merging.

large number of versions, and in our current system, versions are too expensive to allow this, but we are pursuing more efficient version management in our next design that might allow this approach.

3. Detect and break cycles. This is our current solution.

Before adding an in-memory provenance record, the collector checks the in-memory ancestry graph for cycles. If the new record would create a cycle, the collector invokes a cycle-breaking algorithm.

The simplest way to break a cycle is to start a new version for the object that would create the cycle. (If a process were about to read from a file that was its own descendant, we would create a new version for the process; if a process were about to write a file that was its own ancestor, we would create a new version for the file.) This solution has two problems. First, the target might be a file, and creating extraneous versions of files is undesirable. Second, when data flows in a loop, it is often because of a circular operation (e.g., recompiling your compiler) and it is likely to repeat more than once. Creating a new version on every iteration causes a form of provenance explosion that we refer to as *version pumping*.

We developed *node-merging* to avoid these problems. Node-merging treats a set of processes that cycle data among themselves as a single entity for provenance purposes. The provenance of this shared entity is the union of the provenance of the processes that form it. Figure 2 illustrates this approach.

Specifically, if the cycle contains processes P_i and files F_j , the collector starts a new version for each P_i , and merges all those new versions into a single “process” P_{new} . The files F_j are then made descendants of P_{new} .

Implementation Details The implementation of the collector is approximately 4850 lines of code, plus approximately 250 lines of code to intercept system calls.

Each provenanced object within the kernel points to a “virtual vnode” structure (vpnode). This structure holds the in-memory provenance records and the graph edges used for cycle detection. Both persistent and non-persistent objects (such as pipes) use the same data structures.

Capturing provenance for data flowing through pipes requires no special effort: a pipe has an inode structure in the kernel just like a file, and this has a vpnode attached to it. Writing to a pipe creates an ancestry reference from the pipe to the source process; reading the data out again creates an ancestry reference from the sink process back to the pipe. Since the pipe is non-persistent, these references are traversed when provenance is written. Entire pipelines can be captured this way.

The collector communicates with the storage layer via five new VFS calls:

1. `getpnid` - returns a file’s on-disk pnode number
2. `thaw` - begin a new version of a file
3. `freeze` - end a file version
4. `write_prov_string` - transforms a non-reference in-memory provenance record into a persistent provenance record
5. `write_prov_xref` - transform an in-memory file reference into a persistent provenance record

5.3 The Storage Layer

The storage layer is composed of a stackable file system, called PASTA, that uses an in-kernel database engine to store its meta-data. Pasta uses the FiST [30] toolkit for Linux to layer itself on top of any conventional file system. We use `ext2fs` as our underlying file system.

We use an in-kernel port of the Berkeley DB embedded database library [19], called KBDB [14], to store and index provenance. Berkeley DB provides fast, indexed lookup and storage for key/value pairs. Berkeley DB does not provide a schema in the traditional sense. Instead, applications define their schema by creating indexed databases (tables in relational parlance), secondary indexes, and relationships between the tables. An entry in a Berkeley DB database consists of an indexed key and an opaque data item. The application determines how data items are interpreted.

We store provenance in five Berkeley DB databases, summarized in Table 3. The PROVENANCE database is the primary repository of provenance records. The record types stored in the PROVENANCE database are shown in Table 2. The MAP database maps inode numbers to pnode numbers. A file’s pnode number changes

Database	Key	Values
MAP	inode number	pnode number
PROVENANCE	pnode number, record type	provenance data
ARGDATA	sequence number	command line text
ARGREVERSE	command line text	sequence number
ARGINDEX	words from <code>argv</code>	pnode number

Table 3: Provenance databases.

each time it is *thawed*. Command lines and environments are large relative to other provenance and are often repeated. The ARGDATA database implements an optimization, assigning (via Berkeley DB) a sequence number to each unique command line and environment. The ARGREVERSE database maps instances of an environment or command line to the sequence number assigned by ARGDATA. The ARGINDEX database is another secondary index, mapping components of command lines and environments (e.g., individual environment variables and their values) to pnode numbers.

User annotations to a file are stored in the PROVENANCE database with record type ANNOTATION. We provide an annotation `ioctl` that is used to record annotations to a file. This `ioctl` takes the annotation name and a string value as input and records them in the database. Annotations do not pass through the collector and are stored directly into the database.

5.4 The Query Tools

We make the provenance database accessible to users as a standard set of Berkeley DB databases. Making the provenance accessible as standard Berkeley DB files provides many advantages. We can use a variety of programming languages for building query tools (e.g., Python, Perl, Tcl, Java, C), and the standard Berkeley DB utilities (e.g., `dump`, `load`) also work.

As we discuss in greater detail in Section 7.1, our prototype does not meet the security requirements identified earlier, but it does allow us to gain experience with PASS, deploy it in constrained situations, and extract as much information as possible to drive development of our later prototypes.

We built an easy-to-use query tool atop the Berkeley DB databases. The query tool has a built-in file system browser, the Provenance Explorer. Users navigate to the file of interest and run queries using the Provenance Explorer. The MAKEFILE GENERATION query generates a set of commands corresponding to the sequence of events that led to the file's current state. Users can set various filters to, for example, remove commands that occurred before or after a given point in time. Another query,

Query	Functionality	Use cases
MAKEFILE GENERATION	show script to recreate a file	2.1, 2.4, 2.5
DUMP ALL	show all provenance records for a file	2.2, 2.6, 2.3
COMMAND-LINE LOOKUP	find files modified by a process given matching command line arguments	2.4
LIST ANNOTATIONS	show all user-defined annotations to a file	-

Table 4: Query tool summary. The last column maps the query to the example use cases mentioned in Section 2.

DUMP ALL, retrieves the complete provenance for a selected file. The Provenance Explorer also allows users to add or retrieve file annotations. We also support the command-line argument lookup query that allows users to search for files based on arguments to the processes that modified them. For example, a computational physicist can look up files that were modified by a process with an argument, `particle`, set to a particular value. The query capabilities are summarized in Table 4.

The Provenance Explorer is written in Java and uses JNI to access the Berkeley DB databases. To facilitate scripting, we also built command-line utilities that provide functionality equivalent to the Provenance Explorer.

5.5 Prototype versus Requirements

Our prototype has been a source of insight and future challenges despite its limitations. We now analyze these limitations in light of the vision we outlined in Section 4.

The prototype accurately collects and maintains provenance. The provenance it collects for computations that are encapsulated in a process is complete. To the best of our knowledge, this is the first system that captures such information automatically and makes it generally accessible. Unlike many existing systems, it does so without requiring *a priori* specification of the workflow or DAG.

Although we create reasonable provenance for files copied from non-PASS file systems on the local machine, we do not yet create provenance for files that are transmitted across the net. Users and applications can add annotations and application-specific provenance, but for the long-term, we will implement network adapters that observe network activity and create provenanced in-memory objects corresponding to the networked object.

A complete provenance solution requires provenance-aware applications, which are not yet commonly available. However, as we demonstrated in Section 2, the functionality we currently provide does not exist in cur-

rent systems and is complementary to domain-specific solutions. There are existing APIs for provenance-aware applications [11], and the interfaces we have defined for kernel provenance collection offer an alternative. Making an application provenance-aware requires that a programmer identify and then express the dependencies between the objects that an application manipulates. Given the existence of provenance-aware environments [10] and the level of activity in this area, we do not anticipate significant obstacles. Building several provenance-aware applications is part of our next generation design process and will undoubtedly inform the future API.

We have not yet implemented security. This functionality was not important to our first generation users. We wholeheartedly believe that we cannot add security to an already existing system, so we are intentionally designing our next generation PASS from scratch, incorporating what we have learned about provenance collection as well as what we have ascertained about appropriate security models for PASS [3].

The prototype provides simple query capabilities. These capabilities can be extended in a number of ways, but we provide the critical capabilities that we and our users identified. Our current script generation is primitive (it cannot reproduce pipelines without incurring huge overheads), but does meet the needs of our current users.

6 Evaluation

PASS requires in-kernel processing to track provenance, and it produces additional data that is ultimately written to disk. We describe our evaluation platform and then present benchmark results illustrating both the disk-space and time overheads of our prototype. We then present query times for some sample queries run on the output of a Linux build. Finally, we present a sample application and its overhead.

6.1 Evaluation Platform

We evaluated our PASS prototype on a 500Mhz Pentium III machine with 768MB of RAM, running Redhat 7.3. We ran all experiments on a 40GB 7200 RPM Maxtor DiamondMax Plus 8 ATA disk drive. To quantify the overhead of our system, we took measurements on both a PASS and a non-PASS system. We obtain results marked "PASS" by running our modified Linux 2.4.29 kernel (supporting provenance collection) and our provenance-aware file system, PASTA, running on top of ext2fs. We obtain non-PASS results, marked "Ext2", running on an unmodified Linux 2.4.29 kernel and ext2fs, without PASTA.

To ensure a cold cache, we reformatted the file system on which the experiments took place between test runs. Creating the five provenance databases on a newly reformatted file system introduces 160 KB space overhead, which we consider negligible, given today's enormous file systems. We used Auto-pilot [28] to run all our benchmarks. For all experiments, we measured system, user, and elapsed times, as well as the amount of disk space used for provenance. We compute wait time, which usually represents I/O time, as the difference between elapsed time and the sum of system and user times. We ran each experiment 10 times. In nearly all cases, the standard deviations were less than 5%. We do not discuss user time as our code is in the kernel and does not affect the user time.

6.2 Benchmark Performance

We begin with the large and small file microbenchmarks frequently used to evaluate file system performance [24]. These benchmarks exercise the `creat`, `read`, `write`, `fsync`, and `unlink` system calls for a large number of files spread over a directory hierarchy.

Phase	Ext2	PASS	% Overhead
create	10420	12180	16.89
read	10420	12180	16.89
write	10420	13476	29.33
write-sync	10420	13892	33.32
delete	24	3486	14,466

Table 5: Space overhead (in KB) for the small file microbenchmark.

Phase	Ext2	PASS	% Overhead
create	0.91	3.02	232.41
read	0.71	1.51	111.90
write	1.62	4.42	172.09
write-sync	1.67	5.31	217.44
delete	1.13	1.22	7.56

Table 6: Time overhead (in seconds) for the small file microbenchmark.

The small file test uses 2500 4 KB files with 25 files per directory. Table 5 and 6 show that the overheads can be large, but are acceptable for two reasons: First, the absolute numbers are also quite small, so expressing the overhead as a percentage is misleading. Second, this is a particularly challenging benchmark for PASS, because the files are small and the write phases of the benchmark

overwrite existing files. (Only the create phase writes new files.) The overhead during the read phase is due to duplicate elimination done by a PASS. Since the benchmarking process has as many as 2,500 files open, checking for duplicates introduces significant overhead. The overheads during the create, write and write-sync phases are higher than during the read phase, because they generate provenance records in addition to detecting duplicates. The overhead during the delete phase is due to the deletion of entries in the MAP database. The total data size remains unchanged while the provenance grows, suggesting that *provenance pruning*, discussed in Section 7.1, is an important area for further research.

Phase	Ext2	PASS	% Overhead
seq-write	6.16	7.11	15.47
seq-read	0.90	0.92	2.01
rand-write	6.20	7.04	13.58
rand-read	0.90	0.92	2.31
re-read	0.89	0.92	2.44

Table 7: Time overhead (in seconds) for the large file microbenchmark.

The large file benchmark consists of five phases. First, it creates a 100MB file by sequentially writing in units of 256KB; second, it reads the file sequentially; third, it writes 100MB in 256 KB units to random locations in the existing file; fourth, it reads 100MB in 256KB units from random locations in the file; last, it re-reads the file again sequentially. The benchmark has virtually no space overhead, since it uses one file and adds only 10 new records to the provenance databases.

Table 7 shows the time overhead. As a proportion of the running time, these are much better than the small file results, because the “application” is doing more work and the provenance collector proportionally less. The write overheads are due to the stackable file system component of our storage layer. As shown in the next sections the large file results are more indicative of the overheads on realistic workloads.

	Makefile	Dump All
elapsed time	1055.09	91.90
system time	190.98	26.37
user time	817.12	17.57
average elapsed time per file	0.065	0.006
average db lookups per file	485	36

Table 8: Query times, in seconds, for all files present after a Linux build.

For our last benchmark, we built the vanilla Linux 2.4.29 kernel. This build generates 399 MB of data producing 11% space overhead and 10.5% execution time overhead (increasing from 2227 seconds to 2461 seconds). The time overheads are due to the collector and database computational overheads and the I/O time to write the provenance to disk.

We then used the directory tree and files resulting from the Linux compile to measure provenance query times. For each file existing on PASS after the Linux build, we issued two queries: a makefile generation query and a dump_all query. The makefile generation query retrieves records from the PROVENANCE database. The dump_all query retrieves records from the PROVENANCE and ARGDATA databases. The PROVENANCE database had 472,356 records and the ARGDATA database had 5,189 records. The makefile generation query benchmark retrieves 7,911,886 records in all and the maximum ancestor depth is 15. The dump_all query benchmark retrieves 587,494 records in all. Table 8 shows that the queries are fast: even though a makefile query performs nearly 500 lookups per file, it does so in only 65 ms due to the index capabilities of Berkeley DB. However, the writes to these indices contribute to the high write overheads in the small-file benchmark.

Next, we used the same tree to demonstrate how provenance grows as files are modified. We chose ‘N’ files randomly, appending a comment to each one, and then rebuilt the kernel. Table 9 shows how the provenance grows for different values of ‘N’.

N	Size (MB)	% Files changed	% Space growth	% Record growth
0	44	-	-	-
10	44	0.06	0	0.63
110	44	0.67	0	2.31
310	45	1.90	2.27	3.89
1310	49	8.03	11.36	17.18
3310	52	20.28	18.18	34.73
8310	61	50.93	38.64	72.16

Table 9: Provenance Growth benchmark results. where N is the number of files modified and size is the total provenance size after the compile.

The results indicate that provenance growth rate is reasonable considering that changes to the files are small but PASS still has to enter the same number of records into its database irrespective of whether the changes are small or large.

6.3 Application Performance

One of our early users was a computational biologist who regularly uses `blast` [1] to find the protein sequences in one species that are closely related to the protein sequences in another species. Typically, she starts with two files containing the protein sequences of two different species of bacteria. A program, `formatdb`, formats the files and prepares them for `blast`. Then she runs `blast` followed by a series of Perl scripts to produce a list of the proteins in the two species that might be related to each other evolutionarily. When the output is satisfactory, she uses the PASS query tools to generate a script containing the precise sequence of commands that produced the output file.

The time overhead for this real-world workload is minimal, 1.65% (from 309 seconds to 315 seconds), so the new features incur no perceptible cost.

6.4 Evaluation Summary

PASS provides functionality unavailable in other systems with moderate overhead. With the exception of the small file microbenchmark, our time and space overheads are small – typically less than 10% and always under 20%. We and our users are satisfied with these overheads. In addition, we found the system was useful in unanticipated ways such as its ability to generate scripts and its ability to uncover system mysteries.

7 Conclusion

Provenance-aware storage is a new research area that exposes myriad research challenges that we outline in the following section.

7.1 PASS Research Challenges

As we developed our prototype, we encountered problems with our initial design as well as problems that the design did not anticipate. Despite flaws in the prototype, we continued development so we could learn as much as possible before beginning the design of a second generation system. The current prototype has been deployed to a few early adopters, but is not yet in heavy use. We are designing a second prototype based on the experience we have gained. In the following sections, we outline what we have learned and where significant future research challenges remain.

We began the PASS implementation with the simplest and lowest-level schema that could meet our query needs. In parallel with development of the prototype, we undertook an evaluation comparing different provenance storage solutions: our existing one, a relational data

model, an XML-based representation, and an LDAP-based representation [13]. Results so far suggest that the Berkeley DB-based implementation provides superior performance, but these results are not yet final [17].

We recognized the security challenges early in the project and that they are a source of myriad research opportunities. We conducted a low fidelity pilot user study to gain insight into this area and identified the two-pronged security model that separately addresses provenance ancestry and attributes.

The cycle-breaking algorithm we described in Section 5.2 is complicated and has been the single greatest source of errors in the system. Cycle-free provenance collection is an explicit goal for our next prototype, but this remains a research challenge.

As currently implemented, the provenance databases are append-only. This is dangerous, wasteful, and not viable in the long-term. Some provenance pruning is easy: deleting a file with no descendants should delete its provenance. However, deleting items with descendants may offer subtle opportunities for pruning, for example, by compressing long chains of pnodes into equivalent single pnodes. Tackling the general provenance pruning problem requires synthesizing research in information flow and garbage collection and applying this work to this new domain.

Provenance-aware storage will be more broadly accepted once we demonstrate integration with existing provenance solutions. Building provenance-aware applications is the first step in this direction. Discussions with users suggests that building a provenance-aware R environment [22] will make the platform attractive to biologists and social scientists. This is next on our agenda, and we hope that others will select their favorite tools to make provenance-aware.

Until all storage systems are provenance-capable, we face interoperability challenges. Moving files through a non-PASS system should not lose provenance, although it may prove difficult. Extended attributes have been in use for many years, yet there is still no safe way to move files and their extended attributes among systems with and without extended attribute support. Ultimately, we must develop provenance-aware network protocols so provenance can be atomically transmitted with data.

Finally, there remain difficult engineering issues. Our current handling of `mmap` is primitive and needs improvement. The `ptrunc` utility mentioned earlier is equally primitive; we prefer, instead, to allow a user or administrator to designate files (e.g., `/etc/mstab`) that should be ignored in provenance.

We built our PASS prototype to facilitate rapid implementation and deployment, but requiring a particular operating system is not a long-term solution. Instead, we need to develop network-attached PASS imple-

mentations complete with client-side plug-ins for NFS and CIFS clients. We will also build a versioning provenance-aware file system; exploring the considerations involved is another open research problem.

7.2 Acknowledgements

We thank our corporate sponsors, Network Appliance and IBM, for supporting this work. We also thank those who reviewed our paper and gave us excellent feedback, particularly our shepherd, Jason Flinn, for repeated careful and thoughtful reviews.

7.3 In Closing ...

We presented provenance management as a task for storage systems and described and evaluated our prototype system that addresses this problem. It provides a substrate offering significant provenance functionality and lends itself to unifying system and application provenance. We described several use cases where system provenance provides new capabilities and demonstrated that we can accomplish it with acceptable overhead.

References

- [1] ALTSCHUL, S. F., GISH, W., MILLER, W., MYERS, E. W., AND LIPMAN, D. J. Basic local alignment search tool. *Molecular Biology* 215 (1990), 403–410.
- [2] BARU, C., MOORE, R., RAJASEKAR, A., AND WAN, M. SDSC Storage Resource Broker. In *CASCON* (Toronto, Canada, Nov–Dec 1998).
- [3] BRAUN, U., AND SHINNAR, A. A Security Model for Provenance. Technical Report TR-04-06, Harvard University, Jan. 2006.
- [4] BUNEMAN, P., KHANNA, S., AND TAN, W. Why and Where: A Characterization of Data Provenance. In *International Conference on Database Theory* (London, UK, Jan. 2001).
- [5] ClearCase. <http://www.ibm.org/software/awdtools/clearcase>.
- [6] DEELMAN, E., SINGH, G., ATKINSON, M. P., CHERVENAK, A., HONG, N. P. C., KESSELMAN, C., PATIL, S., PEARLMAN, L., AND SU, M.-H. Grid-Based Metadata Services. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM04)* (June 2004).
- [7] FOSTER, I., AND KESSELMAN, C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing* (Summer 1997).
- [8] FOSTER, I., VOECKLER, J., WILDE, M., AND ZHAO, Y. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *CIDR* (Asilomar, CA, Jan. 2003).
- [9] FREW, J., AND BOSE, R. Earth system science workbench: A data management infrastructure for earth science products. In *Proceedings of the 13th International Conference on Scientific and Statistical Database Management* (2001), IEEE Computer Society, pp. 180–189.
- [10] GenePattern. <http://www.broad.mit.edu/cancer/software/genepattern>.
- [11] GROTH, P., MOREAU, L., AND LUCK, M. Formalising a protocol for recording provenance in grids. In *Proceedings of the UK OST e-Science Third All Hands Meeting* (Nottingham, UK, Sept. 2004).
- [12] HEYDON, A., LEVIN, R., MANN, T., AND YU, Y. The Vesta Approach to Software Configuration Management. Technical Report 168, Compaq Systems Research Center, March 2001.
- [13] HODGES, J., AND MORGAN, R. Lightweight Directory Access Protocol (v3). <http://www.rfc-editor.org/rfc/rfc3377.txt>, Sept. 2002.
- [14] KASHYAP, A. File System Extensibility and Reliability Using an in-Kernel Database. Master's thesis, Stony Brook University, December 2004. Technical Report FSL-04-06.
- [15] Lineage File System. <http://crypto.stanford.edu/~cao/lineage.html>.
- [16] MOORE, J. S. The txdit package — interlisp text editing primitives. Manual CSL-81-2, XEROX PARC, Jan. 1981.
- [17] MUNISWAMY-REDDY, K.-K. Deciding How to Store Provenance. Technical Report TR-03-06, Harvard University, Jan. 2006.
- [18] NOST. Definition of the flexible image transport system (FITS), 1999.
- [19] OLSON, M. A., BOSTIC, K., AND SELTZER, M. I. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track* (Monterey, CA, June 1999).
- [20] PANCERELLA, C., ET AL. Metadata in the Collaboratory for Multi-scale Chemical Science. In *Dublin Core Conference* (Seattle, WA, 2003).
- [21] Data Dictionary for Preservation Metadata. <http://www.oclc.org/research/projects/pmwg/premis-final.pdf>, May 2005.
- [22] The R Project for Statistical Computing. <http://www.r-project.org>.
- [23] Spatial data transfer standards (SDTS).
- [24] SELTZER, M., SMITH, K., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File System Logging versus Clustering: A Performance Evaluation. In *Winter USENIX Technical Conference* (New Orleans, LA, January 1995).
- [25] SINGH, G., BHARATHI, S., CHERVENAK, A., DEELMAN, E., KESSELMAN, C., MANOHAR, M., PATIL, S., AND PEARLMAN, L. A Metadata Catalog Service for Data Intensive Applications. In *Proceedings of SC2003 Conference* (November 2003).
- [26] FAQ: Frequently Asked Questions on SRB. <http://www.npaci.edu/dice/srb/faq.html>, June 2004.
- [27] WIDOM, J. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *Conference on Innovative Data Systems Research* (Asilomar, CA, January 2005).
- [28] WRIGHT, C. P., JOUKOV, N., KULKARNI, D., MIRETSKIY, Y., AND ZADOK, E. Auto-pilot: A Platform for System Software Benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track* (Anaheim, CA, April 2005), pp. 175–187.
- [29] YOGESH SIMMHAN, BETH PLAILE, D. G. A Survey of Data Provenance Techniques. Technical Report IUB-CS-TR618, Indiana University, Bloomington, 2005.
- [30] ZADOK, E., BĂDULESCU, I., AND SHENDER, A. Extending file systems using stackable templates. In *USENIX Technical Conference* (Monterey, CA, June 1999).

Thresher: An Efficient Storage Manager for Copy-on-write Snapshots

Liuba Shrira*

Department of Computer Science
Brandeis University
Waltham, MA 02454

Hao Xu

Department of Computer Science
Brandeis University
Waltham, MA 02454

Abstract

A new generation of storage systems exploit decreasing storage costs to allow applications to take snapshots of past states and retain them for long durations. Over time, current snapshot techniques can produce large volumes of snapshots. Indiscriminately keeping all snapshots accessible is impractical, even if raw disk storage is cheap, because administering such large-volume storage is expensive over a long duration. Moreover, not all snapshots are equally valuable. Thresher is a new snapshot storage management system, based on novel copy-on-write snapshot techniques, that is the first to provide applications the ability to discriminate among snapshots efficiently. Valuable snapshots can remain accessible or stored with faster access while less valuable snapshots are discarded or moved off-line. Measurements of the Thresher prototype indicate that the new techniques are efficient and scalable, imposing minimal (4%) performance penalty on expected common workloads.

1 Introduction

A new generation of storage systems exploit decreasing storage costs and efficient versioning techniques to allow applications to take snapshots of past states and retain them for long durations. Snapshot analysis is becoming increasingly important. For example, an ICU monitoring system may analyze the information on patients' past response to treatment.

Over time, current snapshot techniques can produce large volumes of snapshots. Indiscriminately keeping all snapshots accessible is impractical, even if raw disk storage is cheap, because administering such large-volume storage is expensive over a long duration. Moreover, not all snapshots are equally valuable. Some are of value

for a long time, some for a short time. Some may require faster access. For example, a patient monitoring system might retain readings showing an abnormal behavior. Recent snapshots may require faster access than older snapshots.

Current snapshot systems do not provide applications with the ability to *discriminate* efficiently among snapshots, so that valuable snapshots remain accessible while less valuable snapshots are discarded or moved off-line. The problem is that incremental copy-on-write, the basic technique that makes snapshot creation efficient, *entangles* on the disk successive snapshots. Separating entangled snapshots creates disk fragmentation that reduces snapshot system performance over time.

This paper describes Thresher, a new snapshot storage management system, based on a novel snapshot technique, that is the first to provide applications the ability to discriminate among snapshots efficiently. An application provides a discrimination policy that ranks snapshots. The policy can be specified when snapshots are taken, or later, after snapshots have been created. Thresher efficiently disentangles differently ranked snapshots, allowing valuable snapshots to be stored with faster access or to remain accessible for longer, and allowing less-valuable snapshots to be discarded, all without creating disk fragmentation.

Thresher is based on two key innovations. First, a novel technique called *ranked segregation* efficiently separates on disk the states of differently-ranked copy-on-write snapshots, enabling no-copy reclamation without fragmentation. Second, while most snapshot systems rely on a no-overwrite update approach, Thresher relies on a novel update-in-place technique that provides an efficient way to transform snapshot representation as snapshots are created.

The ranked segregation technique can be efficiently composed with different snapshot representations to lower the storage management costs for several useful discrimination policies. When applications need to defer

*This work was supported in part by NSF grant ITR-0428107 and Microsoft.

snapshot discrimination, for example until after examining one or more subsequent snapshots to identify abnormalities, Thresher segregates the normal and abnormal snapshots efficiently by composing ranked segregation with a compact diff-based representation to reduce the cost of copying. For applications that need faster access to recent snapshots, Thresher composes ranked segregation with a dual snapshot representation that is less compact but provides faster access.

A snapshot storage manager, like a garbage collector, must be designed with a concrete system in mind, and must perform well for different application workloads. To explore how the performance of our new techniques depends on the storage system workload, we prototyped Thresher in an experimental snapshot system [12] that allows flexible control of workload parameters. We identified two such parameters, update density and overwriting, as the key parameters that determine the performance of a snapshot storage manager. Measurements of the Thresher prototype indicate that our new techniques are efficient and scalable, imposing minimal (4%) performance penalty on common expected workloads.

2 Specification and context

In this section we specify Thresher, the snapshot storage management system that allows applications to discriminate among snapshots. We describe Thresher in the context of a concrete system but we believe our techniques are more general. Section 3 points out the snapshot system dependent features of Thresher.

Thresher has been designed for a snapshot system called SNAP [12]. SNAP assumes that applications are structured as sequences of *transactions* accessing a storage system. It supports *Back-in-time execution* (or, BITE), a capability of a storage system where applications running general code can run against read-only snapshots in addition to the current state. The snapshots reflect transactionally consistent historical states. An application can choose which snapshots it wants to access so that snapshots can reflect states meaningful to the application. Applications can take snapshots at unlimited “resolution” e.g. after each transaction, without disrupting access to the current state.

Thresher allows applications to discriminate among snapshots by incorporating a snapshot discrimination policy into the following three snapshot operations: a request to take a snapshot (*snapshot request*, or *declaration*) that provides a *discrimination policy*, or indicates lazy discrimination, a request to access a snapshot (*snapshot access*), and a request to specify a discrimination policy for a snapshot (*discrimination request*).

The operations have the following semantics. Informally, an application takes a snapshot by asking for

a snapshot “now”. This snapshot request is serialized along with other transactions and other snapshots. That is, a snapshot reflects all state-modifications by transactions serialized before this request, but does not reflect modifications by transactions serialized after. A snapshot request returns a snapshot *name* that applications can use to refer to this snapshot later, e.g. to specify a discrimination policy for a snapshot. For simplicity, we assume snapshots are assigned unique sequence numbers that correspond to the order in which they occur. A snapshot access request specifies which snapshot an application wants to use for back-in-time execution. The request returns a consistent set of object states, allowing the read-only transaction to run as if it were running against the current storage state. A discrimination policy ranks snapshots. A rank is simply a numeric score assigned to a snapshot. Thresher interprets the ranking to determine the relative lifetimes of snapshots and the relative snapshot access latency.

A snapshot storage management system needs to be efficient and not unduly slow-down the snapshot system.

3 The snapshot system

Thresher is implemented in SNAP [12], the snapshot system that provides snapshots for the Thor [7] object storage system. This section reviews the baseline storage and snapshot systems, using Figure 3 to trace their execution within Thresher.

Our general approach to snapshot discrimination is applicable to snapshot systems that separate snapshots from the current storage system state. Such so-called *split* snapshot systems [16] rely on update-in-place storage and create snapshots by copying out the past states, unlike snapshot systems that rely on no-overwrite storage and do not separate snapshot and current states [13]. Split snapshots are attractive in long-lived systems because they allow creation of high-frequency snapshots without disrupting access to the current state while preserving the on-disk object clustering for the current state [12]. Our approach takes advantage of the separation between snapshot and current states to provide efficient snapshot discrimination. We create a specialized snapshot representation tailored to the discrimination policy while copying out the past states.

3.1 The storage system

Thor has a client/server architecture. Servers provide persistent storage (called database storage) for objects. Clients cache copies of the objects and run applications that interact with the system by making calls to methods of cached objects. Method calls occur within a the context of transaction. A transaction commit causes all

modifications to become persistent, while an abort leaves no transaction changes in the persistent state. The system uses optimistic concurrency control [1]. A client sends its read and write object sets with modified object states to the server when the application asks to commit the transaction. If no conflicts were detected, the server commits the transaction.

An object belongs to a particular server. The object within a server is uniquely identified by an *object reference* (*Oref*). Objects are clustered into 8KB pages. Typically objects are small and there are many of them in a page. An object *Oref* is composed of a *PageID* and a *oid*. The *PageID* identifies the containing page and allows the lookup of an object location using a *page table*. The *oid* is an index into an offset table stored in the page. The offset table contains the object offsets within the page. This indirection allows us to move an object within a page without changing the references to it.

When an object is needed by a transaction, the client fetches the containing page from the server. Only modified objects are shipped back to the server when the transaction commits. Thor provides transaction durability using the ARIES no-force no-steal redo log protocol [5]. Since only modified objects are shipped back at commit time, the server may need to do an *installation read* (*iread*) [8] to obtain the containing page from disk. An in-memory, recoverable cache called the *modified object buffer* (*MOB*) stores the committed modifications allowing to defer *ireads* and increase write absorption [4, 8]. The modifications are propagated to the disk by a background *cleaner* thread that *cleans* the *MOB*. The cleaner processes the *MOB* in transaction log order to facilitate the truncation of the transaction log. For each modified object encountered, it reads the page containing the object from disk (*iread*) if the page is not cached, installs all modifications in the *MOB* for objects in that page, writes the updated page back to disk, and removes the objects from the *MOB*.

The server also manages an in-memory page cache used to serve client fetch requests. Before returning a requested page to the client, the server updates the cache copy, installing all modifications in the *MOB* for that page so that the fetched page reflects the up-to-date committed state. The page cache uses LRU replacement but discards old dirty pages (it depends on *ireads* to read them back during *MOB* cleaning) rather than writing them back to disk immediately. Therefore the cleaner thread is the only component of the system that writes pages to disk.

3.2 Snapshots

SNAP creates snapshots by copying out the past storage system states onto a separate snapshot archive disk.

A snapshot provides the same abstraction as the storage system, consisting of *snapshot pages* and a *snapshot page table*. This allows unmodified application code running in the storage system to run as BITE over a snapshot.

SNAP copies snapshot pages and snapshot page table mappings into the archive during cleaning. It uses an incremental copy-on-write technique specialized for split snapshots: a snapshot page is constructed and copied into the archive when a page on the database disk is about to be overwritten the first time after a snapshot is declared. Archiving a page creates a snapshot page table mapping for the archived page.

Consider the pages of snapshot *v* and page table mappings over the transaction history starting with the snapshot *v* declaration. At the declaration point, all snapshot *v* pages are in the database and all the snapshot *v* page table mappings point to the database. Later, after several update transactions have committed modifications, some of the snapshot *v* pages may have been copied into the archive, while the rest are still in the database. If a page *P* has not been modified since *v* was declared, snapshot page *P* is in the database. If *P* has been modified since *v* was declared, the snapshot *v* version of *P* is in the archive. The snapshot *v* page table mappings track this information, i.e. the archive or database address of each page in snapshot *v*.

Snapshot access. We now describe how BITE of unmodified application code running on a snapshot uses a snapshot page table to look up objects and transparently redirect object references within a snapshot between database and archive pages.

To request a snapshot *v*, a client application sends a *snapshot access request* to the server. The server constructs an archive page table (*APT*) for version *v* (*APT_v*) and “mounts” it for the client. *APT_v* maps each page in snapshot *v* into its archive address or indicates the page is in the database. Once *APT_v* is mounted, the server receiving a page fetch requests from the client looks up pages in *APT_v* and reads them from either archive or database. Since snapshots are accessed read-only, *APT_v* can be shared by all clients mounting snapshot *v*.

Figure 1 shows an example of how unmodified client application code accesses objects in snapshot *v* that includes both archived and database pages. For simplicity, the example assumes a server state where all committed modifications have been already propagated to the database and the archive disk. In the example, client code requests object *y* on page *Q*, the server looks up *Q* in *APT_v*, loads page *Q_v* from the archive and sends it to the client. Later on client code follows a reference from *y* to *x* in the client cache, requesting object *x* in page *P* from the server. The server looks up *P* in *APT_v*

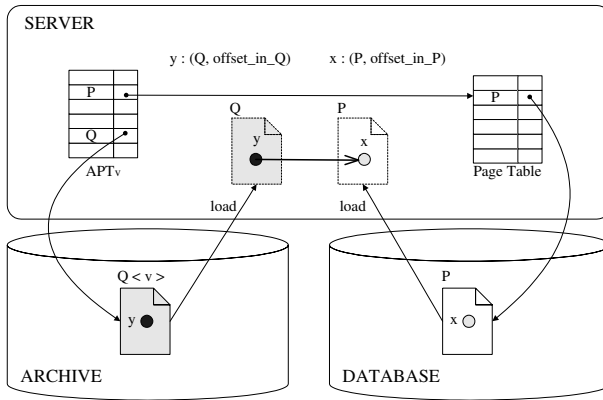


Figure 1: BITE: page-based representation

and finds out that the page P for snapshot v is still in the database. The server reads P from the database and sends it to the client.

In SNAP, the archive representation for a snapshot page includes the complete storage system page. This representation is referred to as *page-based*. The following sections describe different snapshot page representations, specialized to various discrimination policies. For example, a snapshot page can have a more compact representation based on modified object diffs, or it can have two different representations. Such variations in snapshot representation are transparent to the application code running BITE, since the archive read operation reconstructs the snapshot page into storage system representation before sending it to the client.

Snapshot creation. The notions of a *snapshot span* and pages *recorded* by a snapshot capture the incremental copy-on-write manner by which SNAP archives snapshot pages and snapshot page tables. Snapshot declarations partition transaction history into *spans*. The span of a snapshot v starts with its declaration and ends with the declaration of the next snapshot ($v+1$). Consider the first modification of a page P in a span of a snapshot v . The pre-state of P belongs to snapshot v and has to be eventually copied into the archive. We say snapshot v *records* its version of P . In Figure 2, snapshot v records pages P and S (retaining the pre-states modified by transaction tx_2) and the page T (retaining the pre-state modified by transaction tx_3). Note that there is no need to retain the pre-state of page P modified by transaction tx_3 since it is not the first modification of P in the span.

If v does not *record* a version of page P , but P is modified after v is declared, in a span of a later snapshot, the later snapshot records v 's version of P . In above example, v 's version of page Q is recorded by a later snapshot $v+1$ who also records its own version of P .

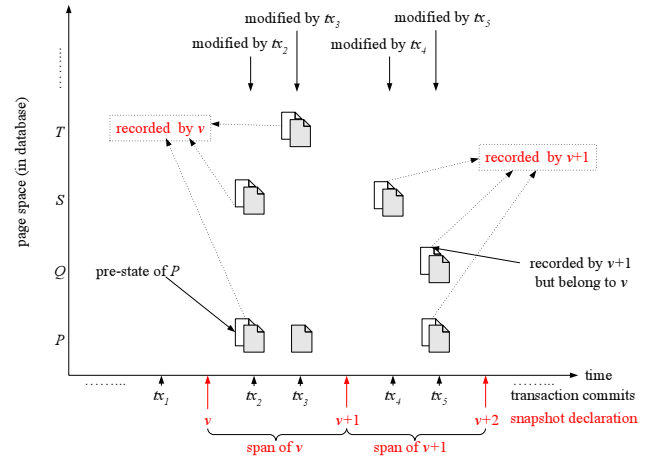


Figure 2: Split copy-on-write

Snapshot pages are constructed and copied into the archive during cleaning when the pre-states of modified pages about to be overwritten in the database are available in memory. Since the cleaner runs asynchronously with the snapshot declaration, the snapshot system needs to prevent snapshot states from being overwritten by the on-going transactions. For example, if several snapshots are declared between two successive cleaning rounds, and a page P gets modified after each snapshot declaration, the snapshot system has to retain a different version of P for each snapshot.

SNAP prevents snapshot state overwriting, without blocking the on-going transactions. It retains the pre-states needed for snapshot creation in an in-memory data structure called *versioned modified object buffer* (VMOB). VMOB contains a queue of buckets, one for each snapshot. Bucket v holds modifications committed in v 's span. As transactions commit modifications, modified objects are added to the bucket of the latest snapshot (Step 1, Figure 3). The declaration of a new snapshot creates a new mutable bucket, and makes the preceding snapshot bucket immutable, preventing the overwriting of the needed snapshot states.

A cleaner updates the database by cleaning the modifications in the VMOB, and in the process of cleaning, constructs the snapshot pages for archiving. Steps 2-5 in Figure 3 trace this process. To clean a page P , the cleaner first obtains a database copy of P . The cleaner then uses P and the modifications in the buckets to create all the needed snapshot versions of P before updating P in the database. Let v be the first bucket containing modifications to P , i.e. snapshot v records its version of P . The cleaner constructs the version of P recorded by v simply by using the database copy of P . The cleaner then updates P by applying modifications in bucket v , removes the modifications from the bucket v , and pro-

ceeds to the following bucket. The updated P will be the version of P recorded by the snapshot that has the next modification to P in its bucket. This process is repeated for all pages with modifications in VMOB, constructing the recorded snapshot pages for the snapshots corresponding to the immutable VMOB buckets.

The cleaner writes the recorded pages into the archive sequentially in snapshot order, thus creating *incremental* snapshots. The mappings for the archived snapshot pages are collected in versioned *incremental* snapshot page tables. VPT_v (versioned page table for snapshot v) is a data structure containing the mappings (from page id to archive address) for the pages recorded by snapshot v . As pages recorded by v are archived, mappings are inserted into VPT_v . After all pages recorded by v have been archived, VPT_v is archived as well.

The cleaner writes the VPTs sequentially, in snapshot order, into a separate archive data structure. This way, a forward sequential scan through the archived incremental page tables from VPT_v and onward finds the mappings for all the archived pages that belong to snapshot v . Namely, the mapping for v 's version of page P is found either in VPT_v , or, if not there, in the VPT of the first subsequently declared snapshot that records P . SNAP efficiently bounds the length of the scan [12]. For brevity, we do not review the bounded scan protocol here.

To construct a snapshot page table for snapshot v for BITE, SNAP needs to identify the snapshot v pages that are in the current database. HAV is an auxiliary data structure that tracks the highest archived version for each page. If $HAV(P) < v$, the snapshot v page P is in the database.

4 Snapshot discrimination

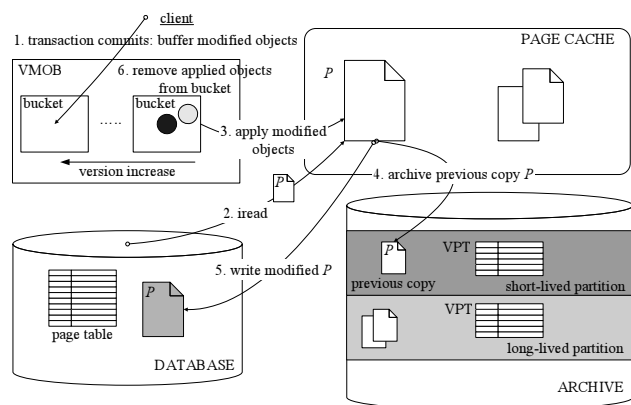


Figure 3: Archiving split snapshots

A snapshot discrimination policy may specify that older snapshots outlive more recently declared snap-

shots. Since snapshots are archived incrementally, managing storage for snapshots according to such a discrimination policy can be costly. Pages that belong to a longer-lived snapshot may be recorded by a later short-lived snapshot thus entangling short-lived and long-lived pages. When different lifetime pages are entangled, discarding shorter-lived pages creates archive storage fragmentation. For example, consider two consecutive snapshots v and $v + 1$ in Figure 2, with v recording pages versions P_v , and S_v , and $v + 1$ recording pages P_{v+1} , Q_{v+1} and S_{v+1} . The page Q_{v+1} recorded by $v + 1$ belongs to both snapshots v and $v + 1$. If the discrimination policy specifies that v is long-lived but $v + 1$ is transient, reclaiming $v + 1$ before v creates disk fragmentation. This is because we need to reclaim P_{v+1} and S_{v+1} but not Q_{v+1} since Q_{v+1} is needed by the long-lived v .

In a long-lived system, disk fragmentation degrades archive performance causing non-sequential archive disk writes. The alternative approach that copies out the pages of the long-lived snapshots, incurs the high cost of random disk reads. But to remain non-disruptive, the snapshot system needs to keep the archiving costs low, i.e. limit the amount of archiving I/O and rely on low-cost sequential archive writes. The challenge is to support snapshot discrimination efficiently.

Our approach exploits the copying of past states in a split snapshot system. When the application provides a snapshot discrimination policy that determines the lifetimes of snapshots, we *segregate* the long-lived and the short-lived snapshot pages and copy different lifetime pages into different archive areas. When no long-lived pages are stored in short-lived areas, reclamation creates no fragmentation. In the example above, if the archive system knows at snapshot $v + 1$ creation time that it is shorter-lived than v , it can store the long-lived snapshot pages P_v , S_v and Q_{v+1} in a long-lived archive area, and the transient P_{v+1} , S_{v+1} pages in a short-lived area, so that the shorter-lived pages can be reclaimed without fragmentation.

Our approach therefore, combines a discrimination policy and a discrimination mechanism. Below we characterize the discrimination policies supported in Thresher. The subsequent sections describe the discrimination mechanisms for different policies.

Discrimination policies. A snapshot discrimination policy conveys to the snapshot storage management system the importance of snapshots so that more important snapshots can have longer lifetimes, or can be stored with faster access. Thresher supports a class of flexible discrimination policies described below using an example. An application specifies a discrimination policy by providing a relative snapshot ranking. Higher-ranked snapshots are deemed more important. By default, every

snapshot is created with a lowest rank. An application can "bump up" the importance of a snapshot by assigning it a higher rank. In a hospital ICU patient database, a policy may assign the lowest rank to snapshots corresponding to minute by minute vital signs monitor readings, a higher rank to the monitor readings that correspond to hourly nurses' checkups, yet a higher rank to the readings viewed in doctors' rounds. Within a given rank level, more recent snapshots are considered more important. The discrimination policy assigns longer lifetimes to more important snapshots, defining a 3-level sliding window hierarchy of snapshot lifetimes.

The above policy is a representative of a general class of discrimination policies we call *rank-tree*. More precisely, a k -level rank-tree policy has the following properties, assuming rank levels are given integer values 1 through k :

- RT1: A snapshot ranked as level i , $i > 1$, corresponds to a snapshot at each lower rank level from 1 to $(i - 1)$.
- RT2: Ranking a snapshot at a higher rank level increases its lifetime.
- RT2: Within a rank level, more recent snapshots outlive older snapshots.

Figure 4 depicts a 3-level rank-tree policy for the hospital example, where snapshot number 1, ranked at level 3, corresponds to a monitor reading that was sent for inspection to both the nurse and the doctor, but snapshot number 4 was only sent to the nurse.

An application can specify a rank-tree policy *eagerly* by providing a snapshot rank at snapshot declaration time, or *lazily*, by providing the rank after declaring a snapshot. An application can also ask to store recent snapshots with faster access. In the hospital example above, the importance and the relative lifetimes of the snapshots associated with routine procedure are likely to be known in advance, so the hospital application can specify a snapshot discrimination policy eagerly.

4.1 Eager ranked segregation

The eager *ranked segregation* protocol provides efficient discrimination for eager rank-tree policies. The protocol assigns a separate archive region to hold the snapshot pages (*volumesⁱ*) and snapshot page tables (VPT^{*i*}) for snapshots at level i . During snapshot creation, the protocol segregates the different lifetime pages and copies them into the corresponding regions. This way, each region contains pages and page tables with the same lifetime and temporal reclamation of snapshots (satisfying policy property RT2) within a region does not create disk fragmentation. Figure 3 shows a segregated archive.

At each rank level i , snapshots ranked at level i are archived in the same incremental manner as in SNAP and at the same low sequential cost. The cost is low because by using sufficiently large write buffers (one for each volume), archiving to multiple volumes can be as efficient as strictly sequential archiving into one volume. Since we expect the rank-tree to be quite shallow the total amount of memory allocated to write buffers is small.

The eager ranked segregation works as follows. The declaration of a snapshot v with a rank specified at level k ($k \geq 1$), creates a separate incremental snapshot page table, VPT _{v} ^{i} for every rank level i ($i \leq k$). The incre-

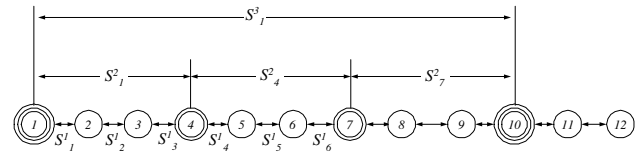


Figure 4: Example rank-tree policy

mental page table VPT _{v} ^{i} collects the mappings for the pages recorded by snapshot v at level i . Since the incremental tables in VPT ^{i} map the pages recorded by *all* the snapshots at level i , the basic snapshot page table reconstruction protocol based on a forward scan through VPT ^{i} (Section 3.2) can be used in region i to reconstruct snapshot tables for level i snapshots.

The recorded pages contain the pre-state before the first page modification in the snapshot span. Since the span for snapshot v at level i (denoted S^i_v) includes the spans of all the lower level snapshots declared during S^i_v , pages recorded by a level i snapshot v are also recorded by some of these lower-ranked snapshots. In Figure 4, the span of snapshot 4 ranked at level 2 includes the spans of snapshots (4), 5 and 6 at level 1. Therefore, a page recorded by the snapshot 4 at level 2 is also recorded by one of the snapshots (4), 5, or 6 at level 1.

A page P recorded by snapshots at multiple levels is archived in the volume of the highest-ranked snapshot that records P . We say that the highest recorder *captures* P . Segregating archived pages this way guarantees that a volume of the shorter-lived snapshots contains no longer-lived pages and therefore temporal reclamation within a volume creates no fragmentation.

The mappings in a snapshot page table VPT _{v} ^{i} in area i point to the pages recorded by snapshot v in whatever area these pages are archived. Snapshot reclamation needs to insure that the snapshot page table mappings are safe, that is, they do not point to reclaimed pages. The segregation protocol guarantees the safety of the snapshot page table mappings by enforcing the following invariant I that constrains the intra-level and inter-level reclamation order for snapshot pages and page tables:

1. VPT_v^i and the pages recorded by snapshot v that are captured in $volume^i$ are reclaimed together, in temporal snapshot order.
2. Pages recorded by snapshot v at level k ($k > 1$), captured in $volume^k$, are reclaimed after the pages recorded by all level i ($i < k$) snapshots declared in the span of snapshot v at level k .

I(1) insures that in each given rank-tree level, the snapshot page table mappings are safe when they point to pages captured in volumes within the same level. I(2) insures that the snapshot page table mappings are safe when they point to pages captured in volumes above their level. Note that the rank-tree policy property RT2 only requires that “bumping up” a lower-ranked snapshot v to level k extends its lifetime but it does not constrain the lifetimes of the lower-level snapshots declared in the span of v at level k . I(2) insures the safety of the snapshot table mappings for these later lower-level snapshots.

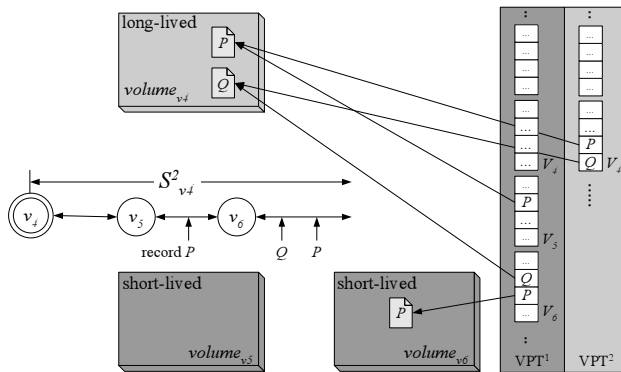


Figure 5: Eager ranked segregation

Figure 5 depicts the eager segregation protocol for a two-level rank-tree policy shown in the figure. Snapshot v_4 , specified at level 2, has a snapshot page table at both level 1 and level 2. The archived page P modified within the span of snapshot v_5 , is recorded by snapshot v_5 , and also by the level 2 snapshot v_4 . This version of P is archived in the volume of the highest recording snapshot (denoted $volume_{v_4}$). The snapshot page tables of both recording snapshots $VPT_{v_5}^1$ and $VPT_{v_4}^2$ contain this mapping for P . Similarly, the pre-state of page Q modified within the span of v_6 is also captured in $volume_{v_4}$. P is modified again within the span of snapshot v_6 . This later version of P is not recorded by snapshot v_4 at level 2 since v_4 has already recorded its version of P . This later version of P is archived in $volume_{v_6}$ and its mapping is inserted into $VPT_{v_6}^1$. Invariant $I(1)$ guarantees that in $VPT_{v_6}^1$ mappings for page P in $volume_{v_6}$ is safe. Invariant $I(2)$ guarantees that in $VPT_{v_6}^1$ the mapping for page Q in volume v_4 is safe.

4.2 Lazy segregation

Some applications may need to defer snapshot ranking to after the snapshot has already been declared (use a *lazy* rank-tree policy). When snapshots are archived first and ranked later, snapshot discrimination can be costly because it requires copying. The lazy segregation protocol provides efficient lazy discrimination by combining two techniques to reduce the cost of copying. First, it uses a more compact diff-based representation for snapshot pages so that there is less to copy. Second, the diff-based representation (as explained below) includes a component that has a page-based snapshot representation. This page-based component is segregated without copying using the eager segregation protocol.

Diff-based snapshots. The compact diff-based representation implements the same abstraction of snapshot pages and snapshot page tables, as the page-based snapshot representation. It is similar to database redo recovery log consisting of sequential repetitions of two types of components, checkpoints and diffs. The checkpoints are incremental page-based snapshots declared periodically by the storage management system. The diffs are versioned page diffs, consisting of versioned object modifications clustered by page. Since typically only a small fraction of objects in a page is modified by a transaction, and moreover, many attributes do not change, we expect the diffs to be compact.

The log repetitions containing the diffs and the checkpoints are archived sequentially, with diffs and checkpoints written into different archive data structures. Like in SNAP, the incremental snapshot page tables collect the archived page mappings for the checkpoint snapshots. A simple page index structure keeps track of page-diffs in each log repetition (the diffs in one log repetition are referred to as *diff extent*).

To create the diff-based representation, the cleaner sorts the diffs in an in-memory buffer, assembling the page-based diffs for the diff extents. The available sorting buffer size determines the length of diff extents. Since frequent checkpoints decrease the compactness of the diff-based representation, to get better compactness, the cleaner may create several diff extents in a single log repetition. Increasing the number of diff extents slows down BITE. This trade-off is similar to the recovery log. For brevity, we omit the details of how the diff-based representation is constructed. The details can be found in [16]. The performance section discusses some of the issues related to the diff-based representation compactness that are relevant to the snapshot storage management performance.

The snapshots declared between checkpoints are re-constructed by first mounting the snapshot page table for

the closest (referred to as *base*) checkpoint and the corresponding diff-page index. This allows BITE to access the checkpoint pages, and the corresponding page-diffs. To reconstruct P_v , the version of P in snapshot v , the server reads page P from the checkpoint, and then reads in order, the diff-pages for P from all the needed diff extents and applies them to the checkpoint P in order. Figure 6 shows an example of reconstructing a page P in

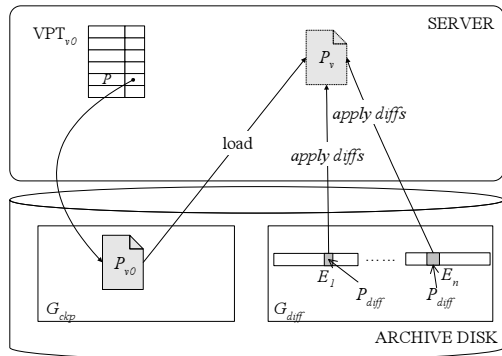


Figure 6: BITE: diff-based representation

a diff-based snapshot v from a checkpoint page P_{v0} and diff-pages contained in several diff extents.

Segregation. When an application eventually provides a snapshot ranking, the system simply reads back the archived diff extents, assembles the diff extents for the longer-lived snapshots, creates the corresponding long-lived base checkpoints, and archives the retained snapshots sequentially into a longer-lived area. If diffs are compact, the cost of copying is low.

The long-lived base checkpoints are created without copying by separating the long-lived and short-lived checkpoint pages using eager segregation. Since checkpoints are simply page-based snapshots declared periodically by the system, the system can derive the ranks for the base checkpoints once the application specifies the snapshot ranks. Knowing ranks at checkpoint declaration time enables eager segregation.

Consider two adjacent log repetitions L_i , L_{i+1} for level-1 snapshots, with corresponding base checkpoints B_i , and B_{i+1} . Suppose the base checkpoint B_{i+1} is to be reclaimed when the adjacent level-1 diff extents are merged into one level 2 diff extent. Declaring the base checkpoint B_i a level-2 *rank tree* snapshot, and base checkpoint B_{i+1} as level-1 *rank tree* snapshot, allows to reclaim the pages of B_{i+1} without fragmentation or copying.

Figure 7 shows an example eager rank-tree policy for checkpoints in lazy segregation. A representation for level-1 snapshots has the diff extents E_1 , E_2 and E_3 (in the archive region G_{diffs}^1) associated with the base

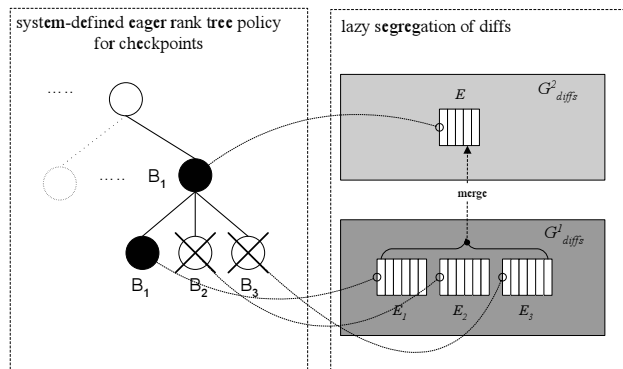


Figure 7: Lazy segregation

checkpoints B_1 , B_2 and B_3 . To create the level-2 snapshots, E_1 , E_2 and E_3 are merged into extent E (in region G_{diffs}^2). This extent E has a base checkpoint B_1 . Eventually, extents E_1 , E_2 , E_3 and checkpoints B_2 , B_3 are reclaimed. Since B_1 was ranked at declaration time as rank-2 longer-lived snapshot, the eager segregation protocol lets B_1 capture all the checkpoint pages it records, allowing to reclaim the shorter-lived pages of B_2 and B_3 without fragmentation.

Our lazy segregation protocol is optimized for the case where the application specifies snapshot rank within a limited time period after snapshot declaration, which we expect to be the common case. If the limit is exceeded, the system reclaims shorter-lived base checkpoints by copying out longer-lived pages at a much higher cost. The same approach can also be used if the application needs to change the discrimination policy.

4.3 Faster BITE

The diff-based representation is more compact but has a slower BITE than the page-based representation. Some applications require lazy discrimination but also need low-latency BITE on a recent window of snapshots. For example, to examine the recent snapshots and identify the ones to be retained. The eager segregation protocol allows efficient composition of diff-based and page-based representations to provide fast BITE on recent snapshots, and lazy snapshot discrimination. The composed representation, called *hybrid*, works as follows. When an application declares a snapshot, hybrid creates two snapshot representations. A page-based representation is created in a separate archive region that maintains a sliding window of W recent snapshots, reclaimed temporally. BITE on snapshots within W runs on the fast page-based representation. In addition, to enable efficient lazy discrimination, hybrid creates for the snapshots a diff-based representation. BITE on snapshots outside W runs on the slower diff-based representation.

Snapshots within W therefore have two representations (page-based and diff-based).

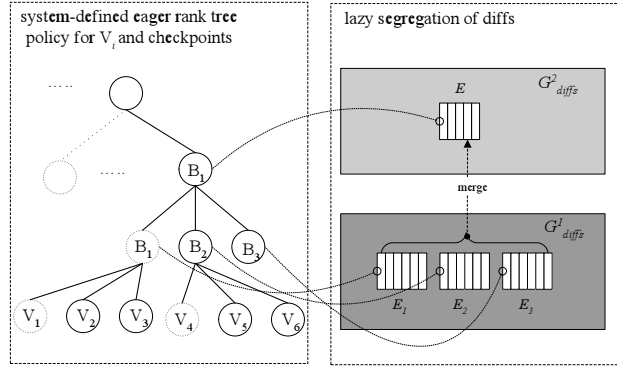


Figure 8: Reclamation in Hybrid

The eager segregation protocol can be used to efficiently compose the two representations and provide efficient reclamation. To achieve the efficient composition, the system specifies an *eager* rank-tree policy that ranks the page-based snapshots as lowest-rank (level-0) rank-tree snapshots, but specifies the ones that correspond to the system-declared checkpoints in the diff-based representation, as level-1. As in lazy segregation, the checkpoints can be further discriminated by bumping up the rank of the longer-lived checkpoints. With such eager policy, the eager segregation protocol can retain the snapshots declared by the system as checkpoints without copying, and can reclaim the aged snapshots in the page-based window W without fragmentation. The cost of checkpoint creation and segregation is completely absorbed into the cost of creating the page-based snapshots, resulting in lower archiving cost than the simple sum of the two representations.

Figure 8 shows reclamation in the hybrid system that adds faster BITE to the snapshots in Figure 7. The system creates the page-based snapshots V_i and uses them to run fast BITE on recent snapshots. Snapshots V_1 and V_4 are used as base checkpoints B_1 and B_2 for the diff-based representation, and checkpoint B_1 is retained as a longer-lived checkpoint. The system specifies an eager rank-tree policy, ranking snapshots V_i at level-0, and bumping up V_1 to level-2 and V_4 to level-1. This allows the eager segregation protocol to create the checkpoints B_1 , and B_2 , and eventually reclaim B_2 , V_5 and V_6 without copying.

5 Performance

Efficient discrimination should not increase significantly the cost of snapshots. We analyze our discrimination techniques under a range of workloads and show they

have minimal impact on the snapshot system performance. Section 6 presents the results of our experiments. This section explains our evaluation approach.

Cost of discrimination. The metric λ_{dp} [12] captures the non-disruptiveness of an I/O-bound storage system. We use this metric to gauge the impact of snapshot discrimination. Let r_{pour} be the “pouring rate” – average object cache (MOB/VMOB) free-space consumption speed due to incoming transaction commits, which insert modified objects. Let r_{drain} be the “draining rate” – the average growth rate of object cache free space produced by MOB/VMOB cleaning. We define:

$$\lambda_{dp} = \frac{r_{drain}}{r_{pour}}$$

λ_{dp} indicates how well the draining keeps up with the pouring. If $\lambda_{dp} \geq 1$, the system operates within its capacity and the foreground transaction performance is not be affected by background cleaning activities. If $\lambda_{dp} < 1$, the system is overloaded, transaction commits eventually block on free object cache space, and clients experience commit delay.

Let t_{clean} be the average cleaning time per dirty database page. Apparently, t_{clean} determines r_{drain} . In Thresher, t_{clean} reflects, in addition to the database ireads and writes, the cost of snapshot creation and snapshot discrimination. Since snapshots are created on a separate disk in parallel with the database cleaning, the cost of snapshot-related activity can be partially “hidden” behind database cleaning. Both the update workload, and the compactness of snapshot representation affect r_{pour} , and determine how much can be hidden, i.e. non-disruptiveness.

Overwriting (α) is an update workload parameter, defined as the percentage of repeated modifications to the same object or page. α affects both r_{pour} and r_{drain} . When overwriting increases, updates cause less cleaning in the storage system because the object cache (MOB/VMOB) absorbs repeated modifications, but high frequency snapshots may need to archive most of the repeated modifications. With less cleaning, it may be harder to hide archiving behind cleaning, so snapshots may become more disruptive. On the other hand, workloads with repeated modifications reduce the amount of copying when lazy discrimination copies diffs. For example, for a two-level discrimination policy that retains one snapshot out of every hundred, of all the repeated modifications to a given object o , archived for the short-lived level-1 snapshots, only one (last) modification gets retained in the level-2 snapshots. To gage the impact of discrimination on the non-disruptiveness, we measure r_{pour} and r_{drain} experimentally in a system with and without discrimination for a range of workloads with

low, medium and high degree of overwriting, and analyze the resulting λ_{dp} .

λ_{dp} determines the maximum throughput of an I/O bound storage system. Measuring the maximum throughput in a system with and without discrimination could provide an end-to-end metric for gauging the impact of discrimination. We focus on λ_{dp} because it allows us to explain better the complex dependency between workload parameters and cost of discrimination.

Compactness of representation. The effectiveness of diff-based representation in reducing copying cost depends on the *compactness* of the representation. We characterize compactness by a relative snapshot retention metric R , defined as the size of snapshot state written into the archive for a given snapshot history length H , relative to the size of the snapshot state for H captured in full snapshot pages. $R = 1$ for the page-based representation. R of the diff-based representation has two contributing components, R_{ckp} for the checkpoints, and R_{diff} for the diffs. *Density* (β), a workload parameter defined as the fraction of the page that gets modified by an update, determines R_{diff} . For example, in a static update workload where any time a page is updated, the same half of the page gets modified, $R_{diff} = 0.5$. R_{ckp} depends on the frequency of checkpoints, determined by L – the number of snapshots declared in the history interval corresponding to one log repetition. In workloads with overwriting, increasing L decreases R_{ckp} since checkpoints are page-based snapshots that record the first pre-state for each page modified in the log repetition. Increasing L by increasing d , the number of diff extents in a log repetition, raises the snapshot page reconstruction cost for BITE. Increasing L without increasing d requires additional server memory for the cleaner to sort diffs when assembling diff pages.

Diff-based representation will not be compact if transactions modify all the objects in a page. Common update workloads have sparse modifications because most applications modify far fewer objects than they read. We determine the compactness of the diff-based representation by measuring R_{diff} and R_{ckp} for workloads with expected medium and low update density.

6 Experimental evaluation

Thresher implements in SNAP [12] the techniques we have described, and also support for recovery during normal operation without the failure recovery procedure. This allows us to evaluate system performance in the absence of failures. Comparing the performance of Thresher and SNAP reveals a meaningful snapshot discrimination cost because SNAP is very efficient: even at

high snapshot frequencies it has low impact on the storage system [12].

Workloads. To study the impact of the workload we use the standard multiuser OO7 benchmark [2] for object storage systems. We omit the benchmark definition for lack of space. An OO7 transaction includes a read-only traversal (T1), or a read-write traversal (T2a or T2b). The traversals T2a and T2b generate workloads with fixed amount of object overwriting and density. We have implemented extended traversal summarized below that allow us to control these parameters. To control the degree of overwriting, we use a variant traversal T2a' [12], that extends T2a to update a randomly selected *AtomicPart* object of a *CompositePart* instead of always modifying the same (*root*) object in T2a. Like T2a, each T2a' traversal modifies 500 objects. The desired amount of overwriting is achieved by adjusting the object update history in a sequence of T2a' traversals. Workload parameter α controls the amount of overwriting. Our experiments use three settings for α , corresponding to low (0.08), medium (0.30) and very high (0.50) degree of overwriting.

To control density, we developed a variant of traversal T2a', called T2f (also modifies 500 objects), that allows to determine β , the average number of modified *AtomicPart* objects on a dirty page when the dirty page is written back to database (on average, a page in OO7 has 27 such objects). Unlike T2a' which modifies one *AtomicPart* in the *CompositePart*, T2f modifies a group of *AtomicPart* objects around the chosen one. Denote by T2f- g the workload with group of size g . T2f-1 is essentially T2a'.

The workload density β is controlled by specifying the size of the group. In addition, since repeated T2f- g traversals update multiple objects on each data page due to write-absorption provided by MOB, T2f- g , like T2a', also controls the overwriting between traversals. We specify the size of the group, and the desired overwriting, and experimentally determine β in the resulting workload. For example, given 2MB of VMOB (the standard configuration in Thor and SNAP for single-client workload), the measured β of multiple T2f-1 is 7.6 (medium α , transaction 50% on private module, 50% on public module). T2f-180 that modifies almost every *AtomicPart* in a module, has $\beta = 26$, yielding almost the highest possible workload density for OO7 benchmark. Our experiments use workloads corresponding to three settings of density β , low (T2f-1, $\beta=7.6$), medium (T2f-26, $\beta=16$) and very high (T2f-180, $\beta=26$) Unless otherwise specified, a medium overwriting rate is being used.

Experimental configuration. We use two experimental system configurations. The single-client experiments run with snapshot frequency 1, declaring a snapshot after each transaction, in a 3-user OO7 database (185MB

in size). The multi-client scalability experiments run with snapshot frequency 10 in a large database (140GB in size). The size of a single private client module is the same in both configurations. All the reported results show the mean of at least three trials with maximum standard deviation at 3%.

The storage system server runs on a Linux (kernel 2.4.20) workstation with dual 64-bit Xeon 3Ghz CPU, 1GB RAM. Two Seagate Cheetah disks (model ST3146707LC, 10000 rpm, 4.7ms avg seek, Ultra320 SCSI) directly attach to the server via LSI Fusion MPT adapter. The database and the archive reside on separate raw hard disks. The implementation uses Linux raw devices and direct I/O to bypass file system cache. The client(s) run on workstations with single P3 850Mhz CPU and 512MB of RAM. The clients and server are inter-connected via a 100Mbps switched network. In single-client experiments, the server is configured with 18 MB of page cache (10% of the database size), and a 2MB MOB in Thor. In multi-client experiments, the server is configured with 30MB of page cache and 8-11MB of MOB in Thor. The snapshot systems are configured with slightly more memory [12] for VMOB so that the same number of dirty database pages is generated in all snapshot systems, normalizing the r_{drain} comparison to Thor.

6.1 Experimental results

We analyze in turn, the performance of eager segregation, lazy segregation, hybrid representation, and BITE under a single-client workload, and then evaluate system scalability under a multiple concurrent client workload.

6.1.1 Snapshot discrimination

Eager segregation. Compared to SNAP, the cost of eager discrimination in Thresher includes the cost of creating VPTs for higher-level snapshots. Table 1 shows t_{clean} in Thresher for a two-level eager rank-tree with inter-level retention fraction f set to one snapshot in 200, 400, 800, and 1600. The t_{clean} in SNAP

Table 1: t_{clean} : eager segregation

f	200	400	800	1600
t_{clean}	5.08ms	5.07ms	5.10ms	5.08ms

is 5.07ms. Not surprisingly, the results show no noticeable change, regardless of retention fraction. The small incremental page tables contribute a very small fraction (0.02% to 0.14%) of the overall archiving cost even for the lowest-level snapshots, rendering eager segregation

essentially free of cost. This result is important, because eager segregation is used to reduce the cost of lazy segregation and hybrid representation.

Lazy segregation. We analyzed the cost of lazy segregation for a 2-level rank-tree by comparing the cleaning costs, and the resulting λ_{dp} in four different system configurations, Thresher with lazily segregated diff-based snapshots (“Lazy”), Thresher with unsegregated diff-based snapshots (“Diff”), page-based (unsegregated) snapshots (“SNAP”), and storage system without snapshots (“Thor”), under workloads with a wide range of density and overwriting parameters. The complete re-

Table 2: Lazy segregation and overwriting

α		t_{clean}	t_{diff}	λ_{dp}
low				
	Lazy	5.30ms	0.13ms	2.24
	Diff	5.28ms	0.08ms	2.26
	SNAP	5.37ms		2.24
	Thor	5.22ms		2.30
medium				
	Lazy	4.98ms	0.15ms	3.67
	Diff	5.02ms	0.10ms	3.69
	SNAP	5.07ms		3.72
	Thor	4.98ms		3.79
high				
	Lazy	4.80ms	0.21ms	4.58
	Diff	4.80ms	0.14ms	4.66
	SNAP	4.87ms		4.61
	Thor	4.61ms		4.83

sults, omitted for lack of space, can be found in [16]. Here we focus on the low and medium overwriting and density parameter values we expect to be more common.

A key factor affecting the cleaning costs in the diff-based systems is the compactness of the diff-based representation. A diff-based system configured with a 4MB sorting buffer, with medium overwriting, has a very low R_{ckp} (0.5% - 2%) for the low density workload (R_{diff} is 0.3%). For medium density workload (R_{diff} is 3.7%), the larger diffs fill sorting buffer faster but R_{ckp} decreases from 10.1% to 4.8% when d increases from 2 to 4 diff extents. These results point to the space saving benefits offered by the diff-based representation.

Table 2 shows the cleaning costs and λ_{dp} for all four systems for medium density workload with low, medium, and high overwriting. The t_{clean} measured in the Lazy and Diff systems includes the database iread and write cost, the CPU cost for processing VMOB, the page

archiving and checkpointing cost via parallel I/O, snapshot page table archiving, and the cost for sorting diffs and creating diff-extents but does not directly include the cost of reading and archiving diffs, since this activity is performed asynchronously with cleaning. The measured t_{diff} reflects these diff related costs (including I/O on diff extents, and diff page index maintenance) per dirty database page. The t_{clean} measured for SNAP and Thor includes the (obvious) relevant cost components.

Compared to Diff, Lazy has a higher t_{diff} reflecting the diff copying overhead. This overhead decreases as overwriting rate increases. t_{diff} does not drop proportionally to the overwriting increase because the dominant cost component of creating higher level extents, reading back the extents in the lowest level, is insensitive to the overwriting rate. Lazy pays no checkpoint segregation cost because it uses the eager protocol.

Next consider non-disruptiveness. We measure r_{pour} and conservatively compute λ_{dp} for Diff and Lazy by adding t_{diff} to t_{clean} , approximating a very busy system where diff I/O is forced to run synchronously with the cleaning. When overwriting is low, λ_{dp} in all snapshot systems is close to Thor. When overwriting is high, all systems have high λ_{dp} because there is very little cleaning in the storage system, and R is low in Diff and Lazy. Importantly, even with the conservative adjustment, λ_{dp} in both diff-based systems is very close to SNAP, while providing significantly more compact snapshots. Notice, all snapshot systems declare snapshots after each traversal transaction. [12] shows that λ_{dp} increases quickly as snapshot frequency decreases.

Hybrid. The Hybrid system incurs the archiving costs of a page-based snapshot system, plus the costs of diff extent creation and segregation, deeming it the costliest of Thresher configurations. Workload density impacts the diff-related costs. Figure 9 shows how the non-disruptiveness λ_{dp} of Hybrid decreases relative to Thor for workloads with low, medium and high density and a fixed medium overwriting. The denser workload implies

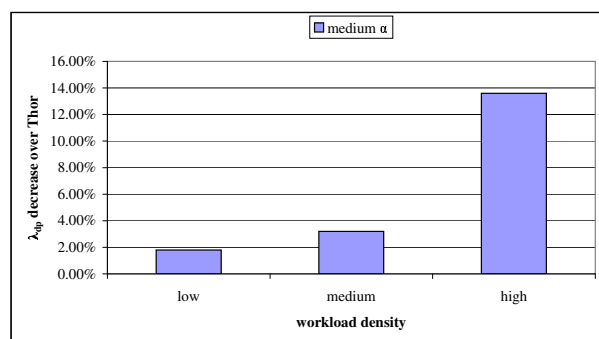


Figure 9: Hybrid: λ_{dp} relative to Thor

more diff I/O. Under the densest possible workload, included here for comparison, the drop of λ_{dp} of hybrid over Thor is 13.6%, where for the common expected medium and low workloads the drop is 3.2% and 1.8% respectively. Note in all configurations, because system's λ_{dp} is greater than 1, there is no client-side performance difference observed between Hybrid and Thor. As a result, the metric λ_{dp} directly reflects the server's "cleaning" speed (t_{clean}). The results in Figure 9 indicate that *Hybrid* is a feasible solution for systems that need fast BITE and lazy discrimination (or snapshot compactness).

6.1.2 Back-in-time execution

We compare BITE in Diff and SNAP to Thor. Our experiment creates Diff and SNAP archives by running 16000 medium density, medium overwriting traversals declaring a snapshot after each traversal. The incremental VPT protocol [12] checkpoints VPTs at 4MB intervals to bound reconstruction scan cost. The APT mounting time, depending on the distance from the VPT checkpoint is of a seesaw pattern, between 21.05ms and 48.77ms. The latency of BITE is determined by the average fetch cost via APT (4.10ms per page).

Diff mounts snapshot by mounting the closest checkpoint, i.e. reconstructing the checkpoint page table (same cost as VPT in SNAP), and mounting the involved page index structures at average mounting time of page index at 7.61ms. To access a page P , Diff reads the checkpoint page and the d diff-pages of P . The average cost to fetch a checkpoint page is 5.80ms, to fetch a diff-page from one extent is 5.42ms. The cost of constructing the requested page version by applying the diff-pages back to the checkpoint page is negligible.

Table 3: End-to-end BITE performance

	current db	page-based	diff-based
T1 traversal	17.53s	27.06s	42.11s

Table 3 shows the average end-to-end BITE cost measured at the client side by running one standard OO7 T1 traversal against Thor, SNAP and Diff respectively. Hybrid has the latency of SNAP for recent snapshots, and latency of Diff otherwise. The end-to-end BITE latency (page fetch cost) increases over time as pages are archived. Table 3 lists the numbers corresponding to a particular point in system execution history with the intention of providing general indication of BITE performance on different representations compared to the performance of accessing the current database. The perfor-

mance gap between page-based and diff-based BITE motivates the hybrid representation.

6.1.3 Scalability

To show the impact of discrimination in a heavily loaded system we compare Thresher (hybrid) and Thor as the storage system load increases, for single-client, 4-client and 8-client loads, for medium density and medium overwriting workload. (An 8-client load saturates the capacity of the storage system).

The database size is 140GB, which virtually contains over 3000 OO7 modules. Each client accesses its private module (45MB in size) in the database. The private modules of the testing clients are evenly allocated within the space of 140GB. Under 1-client, 4-client and 8-client workloads, the λ_{dp} of Thor is 2.85, 1.64 and 1.30 respectively. These λ_{dp} values indicate that Thor is heavily loaded under multi-client workloads. Figure

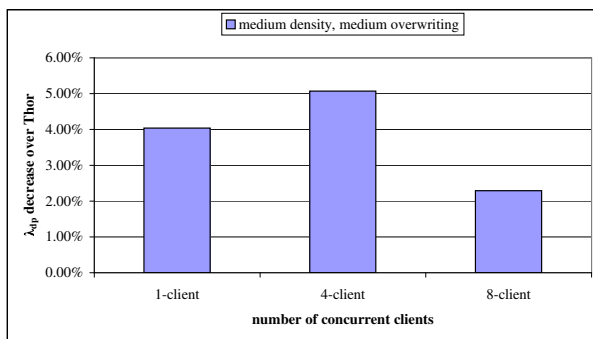


Figure 10: multiple clients: λ_{dp} relative to Thor

10 shows the decrease of λ_{dp} in Hybrid relative to Thor when load increases. Note, that adding more concurrent clients doesn't cause Hybrid to perform worse. In fact, with 8-client concurrent workload, Hybrid performs better than single-client workload. This is because with private modules evenly allocated across the large database, the database random read costs increase compared to the single-client workload, hiding the cost of sequential archiving during cleaning more effectively. Under all concurrent client workloads, Hybrid, the costliest Thresher configurations, is non-disruptive.

7 Related work

Most storage systems that retain snapshots use incremental copy-on-write techniques. To the best of our knowledge none of the earlier systems provide snapshot storage management or snapshot discrimination policies beyond aging or compression.

Versioned storage systems built on top of log-structured file systems and databases [13, 14], and write-anywhere storage [6], provide a low-cost way to retain past state by using no-overwrite updates. These systems do not distinguish between current and past states and use same representation for both. Recent work in ext3cow system [9], separates past and present meta-data states to preserve clustering, but uses no-overwrite updates for data.

Elephant [11] is an early versioned file system that provides consistent snapshots of a file system, allows faster access to recent versions, and provides a sliding window of snapshots but does not support lazy discrimination or different time-scale snapshots.

Compact diff-based representation for versions is used in the CVS source control system. Large-scale storage systems for archiving past state (e.g. [10, 17]) improve the compactness of storage representation (and reduce archiving bandwidth) by eliminating redundant blocks in the archive. These techniques, based on content hashes [10], and differential compression [17], incur high cost at version creation time and do not seem suited for non-disruptive creation of snapshots. However, these systems may benefit from snapshot discrimination.

Generational garbage collectors [15] use efficient storage reclamation techniques that reduce fragmentation by grouping together objects with similar lifetimes. The *rank tree* technique adopts a similar idea for immutable past states shared by snapshots with different lifetimes.

8 Conclusions

We have described new efficient storage management techniques for discriminating copy-on-write snapshots. The ranked segregation technique, borrowing from generational garbage collection, provides no-copy reclamation when the application specifies a snapshot discrimination policy eagerly at snapshot declaration time. Combining ranked segregation with a compact diff-based representation enables efficient reclamation when the application specifies the discrimination policy lazily, after snapshot declaration. Hybrid, an efficient composition of two representations, provides faster access to recent snapshots and supports lazy discrimination at low additional cost.

We have prototyped the new discrimination techniques and evaluated the effect of workload parameters on the efficiency of discrimination. The results indicate that our techniques are very efficient. Eager discrimination incurs no performance penalty. Lazy discrimination incurs a low 3% storage system performance penalty on expected common workloads. The diff-based representation provides more than ten-fold reduction in snapshot storage that can be further reduced with discrimination. Further-

more, the hybrid system that provides lazy discrimination and fast BITE incurs a 10% penalty to the storage system in the worst case of extremely dense update workload, and a low 4% penalty in the expected common case.

Snapshot discrimination could become an attractive feature in future storage systems. The paper has described the first step in this direction. Our prototype is based on a transactional object storage system, although we believe our techniques are more general. We have already applied them to a more general ARIES [5] STEAL system. A file system prototype would be especially worthwhile. It would require modifications to the file system interface along the lines of a recent proposal [3] to enable more efficient capture of updates.

References

- [1] ADYA, A., GRUBER, R., LISKOV, B., AND MAHESHWARI, U. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (1995).
- [2] CAREY, M. J., DEWITT, D. J., AND NAUGHTON, J. F. The OO7 Benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (Washington D.C., May 1993), pp. 12–21.
- [3] DE LOS REYES, A., FROST, C., KOHLER, E., MAMMARELLA, M., AND ZHANG, L. The Kudos Architecture for File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP), WIP Session* (Brighton, UK, October 2005).
- [4] GHEMAWAT, S. *The Modified Object Buffer: A Storage Management Technique for Object-Oriented Databases*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, September 1995.
- [5] GRAY, J. N., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.
- [6] HITZ, D., LAU, J., AND MALCOM, M. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference* (San Francisco, CA, January 1994).
- [7] LISKOV, B., CASTRO, M., SHRIRA, L., AND ADYA, A. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)* (Lisbon, Portugal, June 1999).
- [8] O'TOOLE, J., AND SHRIRA, L. Opportunistic Log: Efficient Installation Reads in a Reliable Storage Server. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, CA, November 1994).
- [9] PETERSON, Z. N., AND BURNS, R. C. The Design, Implementation and Analysis of Metadata for Time Shifting File-system. *Technical Report HSSL-2003-03, Computer Science Department, The John Hopkins University* (Mar. 2003).
- [10] QUINLAN, S., AND DORWARD, S. Venti: A New Approach to Archival Data Storage. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST)* (Monterey, CA, USA, January 2002).
- [11] SANTRY, D., FEELEY, M., HUTCHINSON, N., VEITCH, A., CARTON, R., AND OFIR, J. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)* (Charleston, SC, USA, December 1999).
- [12] SHRIRA, L., AND XU, H. Snap: Efficient snapshots for back-in-time execution. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)* (Tokyo, Japan, Apr. 2005).
- [13] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata Efficiency in Versioning File Systems. In *Proceedings of the 2nd Conference on File and Storage Technologies (FAST)* (San Francisco, CA, USA, March 2003).
- [14] STONEBRAKER, M. The Design of the POSTGRES Storage System. In *Proceedings of the 13th International Conference on Very-Large Data Bases* (Brighton, England, UK, September 1987).
- [15] UNGAR, D., AND JACKSON, F. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems* 14, 1 (Mar. 1992), 1–27.
- [16] XU, H. *Timebox: A High Performance Archive for Split Snapshots*. PhD thesis, Brandeis University, Dec. 2005.
- [17] YOU, L., AND KARAMANOLIS, C. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st IEEE Symposium on Mass Storage Systems and Technologies (MSST)* (College Park, MD, Apr. 2004).

Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines

Partho Nath[†], Michael A. Kozuch^{*}, David R. O'Hallaron[‡], Jan Harkes[‡],
M. Satyanarayanan[‡], Niraj Tolia[‡], and Matt Touns[‡]

[†]*Penn State University*, ^{*}*Intel Research Pittsburgh*, and [‡]*Carnegie Mellon University*

Abstract

This paper analyzes the usage data from a live deployment of an enterprise client management system based on virtual machine (VM) technology. Over a period of seven months, twenty-three volunteers used VM-based computing environments hosted by the system and created over 800 checkpoints of VM state, where each checkpoint included the virtual memory and disk states. Using this data, we study the design tradeoffs in applying content addressable storage (CAS) to such VM-based systems. In particular, we explore the impact on storage requirements and network load of different privacy properties and data granularities in the design of the underlying CAS system. The study clearly demonstrates that relaxing privacy can reduce the resource requirements of the system, and identifies designs that provide reasonable compromises between privacy and resource demands.

1 Introduction

The systems literature of recent years bears witness to a significantly increased interest in virtual machine (VM) technology. Two aspects of this technology, namely platform independence and natural state encapsulation, have enabled the application of this technology in systems designed to improve scalability [6, 14, 16, 32, 40, 49], security [15, 21, 47], reliability [1, 4, 8, 25, 44], and client management [7, 5, 20].

The benefits derived from platform independence and state encapsulation, however, often come with an associated cost, namely the management of significant data volume. For example, enterprise client management systems [7, 20] may require the storage of tens of gigabytes of data *per user*. For each user, these systems store an image of the user's entire VM state, which includes not only the state of the virtual processor and platform devices, but the memory and disk states as well.

While this cost is initially daunting, we would expect a collection of VM state images to have significant data

redundancy because many of the users will employ the same operating systems and applications. Content addressable storage (CAS) [3, 27, 30, 36, 44, 48] is an emerging mechanism that can reduce the costs associated with this volume of data by eliminating such redundancy. Essentially, CAS uses cryptographic hashing techniques to identify data by its *content* rather than by name. Consequently, a CAS-based system will identify sets of identical objects and only store or transmit a single copy even if higher-level logic maintains multiple copies with different names.

To date, however, the benefit of CAS in the context of enterprise-scale systems based on VMs has not been quantified. In this paper, we analyze data obtained from a seven-month, multi-user pilot deployment of a VM-based enterprise client management system called Internet Suspend/Resume (ISR) [19, 37]. Our analysis aims to answer two basic questions:

Q1: By how much can the application of CAS reduce the system's storage requirements?

Q2: By how much can the application of CAS reduce the system's network traffic?

The performance of CAS depends upon several system parameters. The answers to Q1 and Q2, therefore, are analyzed in the context of the two most important of these design criteria:

C1: The *privacy policy*, and

C2: the *object granularity*.

The storage efficiency of a CAS system, or the extent to which redundant data is eliminated, depends upon the degree to which that system is able to *identify* redundant data. Hence, the highest storage efficiency requires users to expose cryptographic digests to the system and potentially to other users. As we shall see, the effects of

this exposure can be reduced but not eliminated. Consequently, criterion C1 represents a trade-off between storage efficiency and privacy.

Object granularity, in contrast, is a parameter that dictates how finely the managed data is subdivided. Because CAS systems exploit redundancy at the object level, large objects (like disk images) are often represented as a sequence of smaller objects. For example, a multi-gigabyte disk image may be represented as a sequence of 128 KB objects (or *chunks*). A finer granularity (smaller chunk-size) will often expose more redundancy than a coarser granularity. However, finer granularities will also require more meta-data to track the correspondingly larger number of objects. Hence, criterion C2 represents the trade-off between efficiency and meta-data overhead.

The results obtained from the ISR pilot deployment indicate that the application of CAS to VM-based management systems is more effective in reducing storage and network resource demands than applying traditional compression technology such as the Lempel-Ziv compression [50] used in *gzip*. This result is especially significant given the non-zero runtime costs of compressing and uncompressing data. In addition, combining CAS and traditional compression reduces the storage and network resource demands by a factor of two beyond the reductions obtained by using traditional compression technology alone.

Further, using this real-world data, we are able to determine that enforcing a strict privacy policy requires approximately 1.5 times the storage resources required by a system with a less strict privacy policy. Finally, we have determined that the efficiency improvements derived from finer object granularity typically outweighs the meta-data overhead. Consequently, the disk image chunksize should be between 4 and 16 KB.

Sections 4 and 5 will elaborate on these results from the pilot deployment. But first, we provide some background on ISR, content addressable storage, and the methodology used in the study.

2 Background

2.1 Internet Suspend/Resume

Internet Suspend/Resume (ISR) is an enterprise client management system that allows users to access their personal computing environments from different physical machines. The system is based on a combination of VM technology and distributed storage. User computing environments are encapsulated by VM instances, and the state of such a VM instance, when idle, is captured by system software and stored on a carefully-managed server. There are a couple of motivations for this idea. First, decoupling the computing environment from the

hardware allows clients to migrate across different hosts. Second, storing VM state on a remote storage repository simplifies the management of large client installations. The physical laptops and desktops in the installation no longer contain any hard user-specific state, and thus client host backups are no longer necessary; the only system that needs to be backed up is the storage repository.

Figure 1 shows the setup of a typical ISR system. The captured states of user environments are known as *parcels* and are stored on a collection of (possibly) distributed *content servers*. For example, in the figure, Bob owns two parcels. One environment includes Linux as the operating system, and the other includes Windows XP.

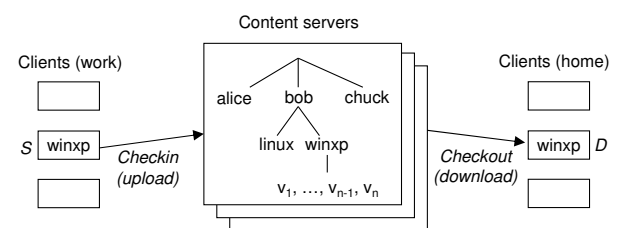


Figure 1: An ISR system.

Each parcel captures the complete state of some VM instance. The two most significant pieces of state are the *memory image* and the *disk image*. In the current ISR deployment, memory images are 256 MB and disk images are 8 GB. Each memory image is represented as a single file. Each disk image is partitioned into a set of 128 KB *chunks* and stored on disk, one file per chunk.

For each parcel, the system maintains a sequence of checkpointed diff-based *versions*, v_1, \dots, v_{n-1}, v_n . Version v_n is a complete copy of the memory and disk image. Each version v_k , $1 \leq v_k \leq v_{n-1}$, has a complete copy of the memory image, along with the chunks from the v_k version of the disk image that changed between version v_k and v_{k+1} .

Each client host in the ISR system runs a *VM monitor* that can load and execute any parcel. ISR provides a mechanism for suspending and transferring the execution of these parcels from one client host to another. For example, Figure 1 shows a scenario where a user transfers the execution of a VM instance from a source host *S* at the office to a destination host *D* at home.

The transfer occurs in two phases: a *checkin* step followed by a *checkout* step. After the user suspends execution of the VM monitor on *S*, the checkin step uploads the memory image and any dirty disk chunks from *S* to one of the content servers, creating a new parcel version on the server. The checkout step downloads the memory image of the most recent parcel version from the content

server to D . The user is then able to resume execution of the parcel on D (even before the entire disk image is present). During execution, ISR fetches any missing disk chunks from the content server *on demand* and caches those chunks at the client for possible later use.

2.2 Content Addressable Storage

Content addressable storage (CAS) is a data management approach that shows promise for improving the efficiency of ISR systems. CAS uses cryptographic hashing to reduce storage requirements by exploiting commonality across multiple data objects [13, 23, 29, 42, 43, 48]. For example, to apply CAS to an ISR system, we would represent each memory and disk image as a sequence of fixed-sized chunk files, where the filename of each chunk is computed using a collision-resistant cryptographic hash function. Since chunks with identical names are assumed to have identical contents, a single chunk on disk can be included in the representations of multiple memory and disk images. The simplest example of this phenomenon is that many memory and disk images contain long strings of zeros, most of which can be represented by a single disk chunk consisting of all zeros. A major goal of this paper is to determine to what extent such redundancy exists in realistic VM instances.

3 Methodology

Sections 4 and 5 present our analysis of CAS technology in the context of ISR based on data collected during the first 7 months of a pilot ISR deployment at Carnegie Mellon University. This section describes the deployment, and how the data was collected and analyzed.

3.1 Pilot Deployment

The pilot deployment (pilot) began in January, 2005, starting with about 5 users and eventually growing to 23 active users. Figure 2 gives the highlights. Users

Number of users	23
Number of parcels	36
User environment	Windows XP or Linux
Memory image size	256 MB
Disk image size	8 GB
Client software	ISR+Linux+VMware
Content server	IBM BladeCenter
Checkins captured	817
Uncompressed size	6.5 TB
Compressed size	0.5 TB

Figure 2: Summary of ISR pilot deployment.

were recruited from the ranks of Carnegie Mellon students and staff and given a choice of a Windows XP parcel, a Linux parcel, or both. Each parcel was configured with an 8 GB virtual disk and 256 MB of memory. The *gold images* used to create new parcels for users were updated at various times over the course of the pilot with security patches.

The content server is an IBM BladeCenter with 9 servers and a 1.5 TB disk array for storing user parcels. Users downloaded and ran their parcels on Linux-based clients running VMware Workstation 4.5.

3.2 Data Collection

During the course of the pilot, users performed numerous checkin operations, eventually creating 817 distinct parcel versions on the content server. In August, 2005, after 7 months of continuous deployment, a snapshot of the memory and disk images of these parcel versions was taken on the content server. In uncompressed form, the snapshot state would have consumed about 6.5 TB. However, due to ISR's diff-based representation and gzip compression, it only required about 0.5 TB of disk space. This snapshot state was copied to another server, where it was post-processed and stored in a database for later analysis.

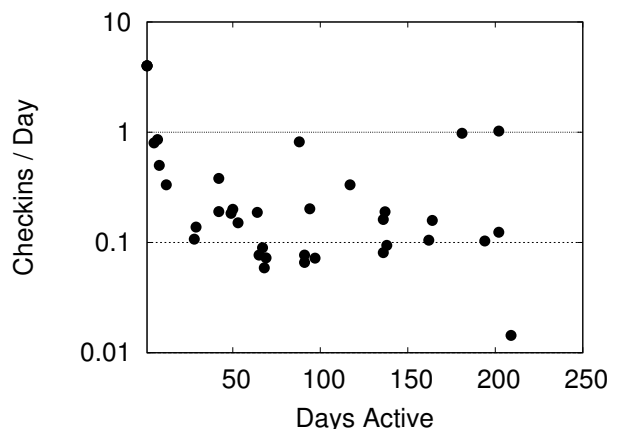


Figure 3: Observed parcel checkin frequency

Figure 3 summarizes parcel usage statistics for the deployment data. Each point in the figure represents a single parcel and indicates the number of days that parcel was active as well as its checkin frequency (average number of checkins per day). Parcels could be active for less than the entire duration of the deployment either because the parcel was created after the initial deployment launch or because a user left the study early (e.g. due to student graduation or end-of-semester constraints). Since new users were added throughout the course of the pilot, during post-processing we normalized the start time of

each user to day zero. No extrapolation of data was performed, thus the usage data for a user who has used the system for n days appears in the first n days worth of data in the corresponding analysis. We also removed several parcels that were used by developers for testing, and thus were not representative of typical use.

3.3 Analysis

The August 2005 snapshot provided a complete history of the memory and disk images produced by users over time. This history allowed us to ask a number of interesting “what if” questions about the impact of different design choices, or policies, on the performance of the ISR system. In particular, we explored three different storage policies: a baseline non-CAS *Delta* policy and two different CAS policies called *IP* and *ALL*. These are summarized in Figure 4. In each approach, a parcel’s

Policy	Encryption	Meta-data
Delta	private per-parcel key	none
IP	private per-parcel key	(tag) array
ALL	convergent encryption	(tag, key) array

Figure 4: Storage policy encryption technique summary.

memory and disk images are partitioned into fixed-sized chunks, which are then encrypted, and optionally compressed using conventional tools like gzip.

As will be shown in sections 4 and 5, differences in the storage and encryption of data chunks affect not only the privacy afforded to users but also dramatically alter the resources required for storage and network transmission. For our evaluations, we chose chunk sizes of 4KB (a typical disk-allocation unit for most operating systems) and larger.

Delta policy. In this non-CAS approach, the most recent disk image v_n contains a complete set of chunks. For each version $k < n$, disk image v_k contains only those chunks that differ in disk image v_{k+1} . Thus, we say that Delta exploits *temporal redundancy* across the versions.

Chunks in all of the versions in a parcel are encrypted using the same per-parcel private key. Individual chunks are addressed by their position in the image (logical block addressing), hence no additional meta-data is needed. Memory images are represented in the same way. Delta is similar to the approach used by the current ISR prototype (the current prototype only chunks the disk image and not the memory image). We chose it as the baseline because it is an effective state-of-the-art non-CAS approach for representing versions of VM images.

IP (intra-parcel) policy. In this CAS approach, each parcel is represented by a separate pool of unique chunks

shared by all versions, v_1, \dots, v_n , of that parcel. Similar to Delta, IP identifies temporal redundancy between contiguous parcel versions. However, IP can also identify temporal redundancy in non-contiguous versions (e.g., disk chunk i is identical in versions 4 and 6, but different in version 5), and it can also identify any *spatial redundancy* within each version.

As with Delta, each chunk is encrypted using a single per-parcel private key. However, each version of each disk image (and each memory image) requires additional meta-data to record the sequence of chunks that comprise the image. In particular, the meta-data for each image is an array of *tags*, where tag i is the SHA-1 hash of chunk i . This array of tags is called a *keyring*.

ALL policy. In this CAS approach, all parcels for all users are represented by a single pool of unique chunks. Each chunk is encrypted using *convergent encryption* [11], where the encryption key is simply the SHA-1 hash of the chunk’s original plaintext contents. This allows chunks to be shared across different parcels and users, since if the original plaintext chunks are identical, then the encrypted chunks will also be identical.

As with IP, each version of each disk image (and each memory image) requires additional keyring meta-data to record the sequence of chunks that compose the image, in this case an array of (tag, key) tuples, where key i is the encryption key for chunk i , and tag i is the SHA-1 hash of the encrypted chunk. Each keyring is then encrypted with a per-parcel private key.

The IP and ALL policies provide an interesting trade-off between privacy and space efficiency. Intuitively, we would expect the ALL policy to be the most space-efficient because it identifies redundancy across the maximum number of chunks. However, this benefit comes at the cost of decreased privacy, both for individual users and the owners/operators of the storage repository. The reason is that ALL requires a consistent encryption scheme such as convergent encryption for all blocks. Thus, individual users are vulnerable to dictionary-based traffic analysis of their requests, either by outside attackers or the administrators of the systems. Owner/operators are vulnerable to similar analysis, if, say, the contents of their repository are subpoenaed by some outside agency.

Choosing appropriate chunk sizes is another interesting policy decision. For a fixed amount of data, there is a tension between chunk size and the amount of storage required. Intuitively, we would expect that smaller chunk sizes would result in more redundancy across chunks, and thus use less space. However, as the chunk size decreases, there are more chunks, and thus there is more keyring meta-data. Other chunking techniques such as Rabin Fingerprinting [26, 31, 38] generate chunks of varying sizes in an attempt to discover redundant data that does not conform to a fixed chunk size. The evalua-

tion of non-fixed-size chunk schemes is beyond the scope of this paper but is on our agenda for future work.

The remainder of the paper uses the data from the ISR deployment to quantify the impact of CAS privacy and chunksize policies on the amount of storage required for the content servers, and the volume of data that must be transferred between clients and content servers.

4 Results: CAS & Storage

Because server storage represents a significant cost in VM-based client management systems, we begin our discussion by investigating the extent to which a CAS-based storage system could reduce the volume of data managed by the server.

4.1 Effect of Privacy Policy on Storage

As expected, storage policy plays a significant role in the efficiency of the data management system. Figure 5 presents the growth in storage requirements over the lifetime of the study for the three different policies using a fixed chunksize (128 KB). As mentioned in Section 3.2, the graph normalizes the starting date of all users to day zero. The growth in the storage from thereon is due to normal usage of disks and storage of memory checkpoints belonging to the users. The storage requirement shown includes both the disk and memory images.

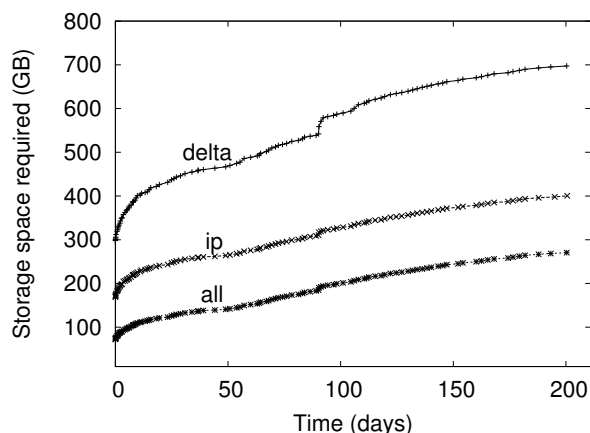


Figure 5: Growth of storage needs for Delta, IP, and ALL.

CAS provides significant savings. As shown in Figure 5, adopting CAS with the IP policy reduces the required server resources at day 201 under the Delta policy by 306 GB, from 717 GB to 411 GB. This reduction represents a savings of 42%.

Recall that adopting CAS is a lossless operation; CAS simply stores the same data more efficiently than the Delta policy. The improved efficiency is due to the fact

that the Delta policy only exploits temporal redundancy between versions. That is, the Delta policy only identifies identical objects when they occur in the same location in subsequent versions of a VM image. The IP policy, in contrast, identifies redundancy anywhere within the parcel – within a version as well as between versions (including between non-subsequent versions).

Note that the 42% space savings was realized without compromising privacy. Users in a CAS-IP-backed system do not expose the contents of their data to any greater degree than users of a Delta-backed system.

Relaxing privacy introduces additional gains. In systems where a small relaxation of privacy guarantees is acceptable, additional savings are possible. When the privacy policy is relaxed from IP to ALL, the system is able to identify additional redundancy that may exist between different users' data. From Figure 5, we see that such a relaxation will reduce the storage resources required by another 133 GB, to 278 GB. The total space savings realized by altering the policy from Delta to ALL is 61%.

On comparing ALL with IP in Figure 5, we see that the curves are approximately parallel to each other. However, under certain situations, a system employing the ALL policy could dramatically outperform a similar system that employs the IP policy. Imagine for example a scenario where a security patch is applied by each of a large number, N , of users in an enterprise. Assuming that the patch affected each user's environment in the same way, by introducing X MB of new data, an IP server would register a total addition of NX MB. In contrast, an ALL server would identify the N copies of the patched data as identical and would consequently register a total addition of X MB.

The starting points of the curves in Figure 5 are also of interest. Because the X-axis has been normalized, this point corresponds to the creation date of all parcels. To create a new parcel account, the system administrator copies a gold image as version 1 of the parcel. Hence, we would assume that the system would exhibit very predictable behavior at time zero.

For example, under the Delta policy which only reduces redundancy *between versions*, the system data should occupy storage equal to the number of users times the space allocated to each user. In the deployment, users were allocated 8 GB for disk space and 256 MB for memory images. Thirty-six parcels should then require approximately 300 GB of storage space which is exactly the figure reported in the figure.

For the IP policy, one would also expect the server to support a separate image for each user. However, CAS had eliminated the redundant data within each of these images yielding an average image size of approximately 4 GB. The observed 171 GB storage space is consistent

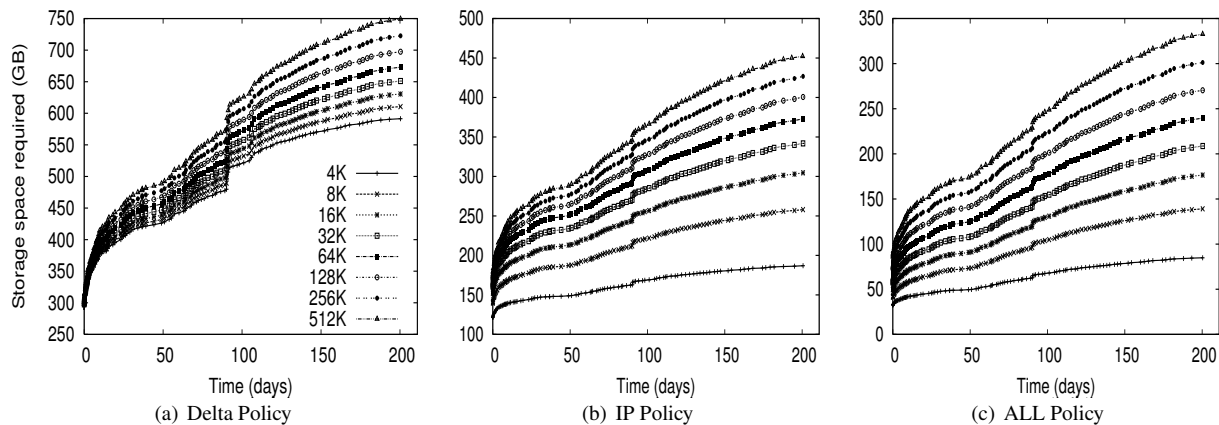


Figure 6: Storage space growth for various chunk sizes without meta-data overhead (y-axis scale varies).

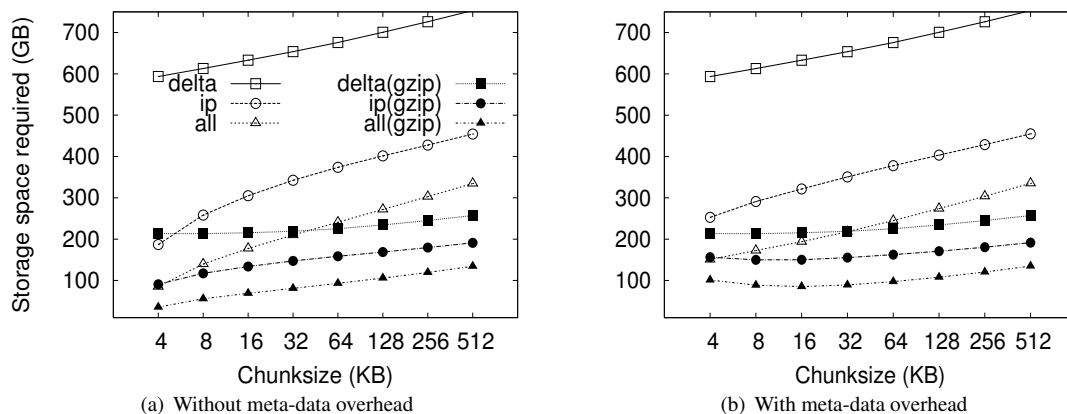


Figure 7: Server space required, after 201 deployment days.

with this expectation.

Under the ALL policy in contrast, one would expect the system to store a single copy of the gold image shared by all users, yielding a total storage requirement of 8 GB plus 256 MB (closer to 4 GB, actually, due to the intra-image redundancy elimination). We were quite surprised, consequently, to observe the 72 GB value reported in the figure. After reviewing the deployment logs, we determined that this value is due to the introduction of multiple gold images into the system. To satisfy different users, the system administrators supported images of several different Linux releases as well as several instances of Windows images. In all, the administrators had introduced 13 different gold images, a number that is consistent with the observed 72 GB of occupied space.

Another point of interest is a disturbance in the curve that occurs at the period around 100 days. We note that the disturbance is significant in the Delta curve, smaller in the IP curve, and almost negligible in the ALL curve. We've isolated the disturbance to a single user and observe that this anomaly is due to the user reorganizing his

disk image without creating new data that did not already exist somewhere in the system. Hence, we conclude that this must have been an activity similar to defragmentation or re-installation of an operating system.

4.2 Effect of Chunksize on Storage

In addition to privacy considerations, the administrator of a VM-based client management system may choose to optimize the system efficiency by tuning the chunksize. The impact of this parameter on storage space requirements is depicted in Figure 6; in this figure, we present what the growth curves of Figure 5 would have been had we chosen different chunk sizes.

Note that the effect of this parameter is not straightforward. Varying the chunksize has three different effects on efficiency.

First, smaller chunk sizes tend to expose more redundancy in the system. As a trivial exercise, consider two objects each of which, in turn, comprises two blocks ($Object_1 = AB$ and $Object_2 = CA$). If the chunksize

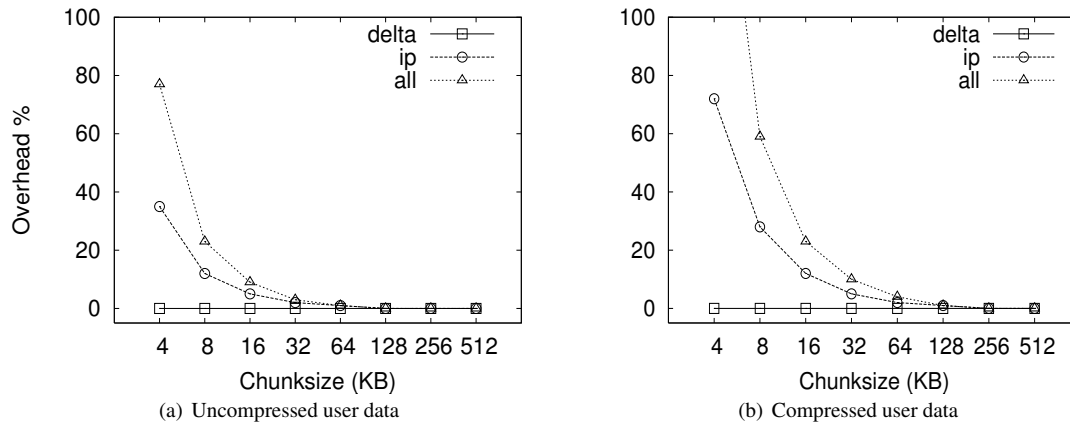


Figure 8: Meta-data overhead expressed as a percentage of user data.

is chosen to be a whole object, the content addresses of *Object₁* and *Object₂* will differ and no redundancy will be exposed. If the chunksize is chosen to be a block, in contrast, the identical *A* blocks will be identified and a space savings of 25% will result.

Second, smaller chunksizes require the maintenance of more meta-data. With the whole-object chunksize from the example above, the system would maintain two content addresses, for *Object₁* and *Object₂*. With the block chunksize, however, the system must maintain two sets of two content addresses so that *Object₁* and *Object₂* may each be properly reconstructed. Note further that this additional meta-data maintenance is required whether or not any redundancy was actually identified in the system.

Third, smaller chunksizes tend to provide a reduced opportunity for post-chunking compression. In addition to chunk-level redundancy elimination through CAS, intra-chunk redundancy may be reduced through traditional compression techniques (such as *gzip*). However, as the chunksize is reduced, these techniques have access to a smaller intra-chunk data pool on which to operate, limiting their efficiency.

To better understand the effect of chunksize, we analyzed the deployment data for all three storage policies with and without compression under several different chunksizes. The results are shown in Figure 7.

All three effects of chunksize can be observed in this figure. For example, Figure 7(a), which ignores the increased meta-data required for smaller chunksizes, clearly indicates that smaller chunksizes expose more redundancy. These gains for small chunk sizes, however, are erased when the meta-data cost is introduced to the storage requirements in Figure 7(b). Finally, the reduced opportunities for compression due to smaller chunksize can be observed in Figure 7(b) by comparing the IP and IP(*gzip*) or ALL and ALL(*gzip*) curves.

CAS is more important than compression. In Figure 7(a), the Delta curve *with compression* intersects the IP and ALL curves *without compression*. The same is true in Figure 7(b) with respect to the ALL curve. This indicates, that given appropriate chunksizes, a CAS-based policy can outperform compression applied to a non-CAS-based policy.

Considering meta-data overheads, the ALL policy outperforms Delta with compression for all the chunksizes less than 64KB. This is a very remarkable result. Compression in the storage layer may be a high latency operation, and it may considerably affect virtual disk operation latencies. By use of CAS, one can achieve savings that exceed traditional compression techniques! If additional space savings are required, compression can be applied after the application of content addressing.

Figure 7(a) shows that compression provides an additional savings of a factor of two to three. For example, the space demands for the ALL policy, drops from 87GB to 36GB when using 4KB chunks, and from 342GB to 137GB when using 512KB chunks.

Exposing redundancy outweighs meta-data overhead. Figure 8 shows the ratio of meta-data (keyring size) to the size of the data. We observe that this ratio is as high as 80% for ALL, and 35% for IP at 4KB chunksize without compression and even higher after compression is applied to the basic data. Yet, from Figure 7(b), we observe from the IP and ALL curves that reducing chunksize always yields a reduction in storage requirements. This indicates that the gains through CAS-based redundancy elimination far exceed the additional meta-data overhead incurred from smaller chunksize.

The picture changes slightly with the introduction of traditional compression. The IP(*gzip*) and ALL(*gzip*) curves of Figure 7(b) indicate that the smallest chunksize is not optimal. In fact, we see from Figure 8 that the meta-data volume becomes comparable to the data

volume at small chunk sizes.

Small chunk sizes improve efficiency. With Figure 7(b), we are in a position to recommend optimal chunk sizes. Without compression, the optimal chunksize is 4 KB for the Delta, IP and ALL policies. With compression, the optimal chunksize is 8 KB for the Delta(gzip) policy and 16 KB for the IP(gzip) and ALL(gzip) policies.

5 Results: CAS & Networking

In a VM-based client management system, the required storage resources, as discussed in the previous section, represent a cost to the system administrator in terms of physical devices, space, cooling, and management. However, certain user operations, such as check-in and checkout, require the transmission of data over the network. While the system administrator must provision the networking infrastructure to handle these transmissions, perhaps the more significant cost is the user time spent waiting for the transmissions to complete.

For example, a common telecommuting scenario may be that a user works at the office for some time, checks-in their new VM state, travels home, and attempts to checkout their VM state to continue working. In the absence of CAS or traditional compression, downloading just the 256 MB memory, which is required before work can resume, over a 1 Mbps DSL line requires more than 30 minutes of wait time. After working at home for some time, the user will also want to checkin their new changes. Because the checkin image is typically larger than the checkout image, and because the upload speed of ADSL is often much slower than the download speed, the checkin operation can often require two hours or more.

Consequently, we devote this section to characterizing the benefits that CAS provides in terms of reducing the volume of data to be transmitted during typical upload (checkin) or download (checkout) operations.

5.1 Effect of Privacy Policy on Networking

As with storage, we begin the discussion by considering the effect of privacy policy on networking. We note that our definition of privacy policy affects the representation of data chunks in storage, not the mechanics of chunk transmission. However, the chosen storage policy can affect the capability of the system to identify redundant data blocks that need not be sent because they already exist at the destination.

As an example, suppose that a user copies a file within their virtual environment. This operation may result in a virtual disk that contains duplicate chunks. Under the IP and ALL policies, at the time of upload, the client

will send a digest of modified chunks to the server, and the server may respond that the duplicate chunks need not be sent because the chunks (identified by the chunks' tags) already exist on the server. Such redundant data can occur for a variety of reasons (particularly under the ALL policy) including the push of software patches, user download of popular Internet content, and the installation and compilation of common software packages.

During download (checkout) operations, the client code will search through the existing version(s) of the user's data on that client to identify chunks that need not be retrieved from the server. As the system is only comparing the latest version on the server with the existing version on the client, the volume of data to be transmitted does not depend on the privacy policy. In contrast, the volume of data transmitted during upload (checkin) operations does depend on the privacy policy employed because, at the server, redundant chunks are only identified within that user's version history under the IP policy, but can be identified across all users' version histories under the ALL policy. These differences based on storage policy are summarized in Figure 9 and affect our discussion in two ways: (1) this section (Section 5.1), which investigates the effects of privacy policy, only considers the upload operation, and (2) Figures 12 and 13 in Section 5.2 contain curves simply labeled CAS that represent the identical download behaviors of the IP and ALL policies.

	Redundancy Comparison	
	Upload (between client copy and...)	Download (between server version N and ...)
Delta	server version N-1	current client version
IP	server versions [1, N-1]	current client version
ALL	all versions/all parcels	current client version

Figure 9: Search space for identifying redundant blocks during data synchronization operations. Note that for download, the system inspects the most recent version available at the client (which may be older than $N - 1$).

CAS is essential. The upload volume for each of the storage policies with and without compression is presented in Figure 10. Because the upload size for any user session includes the 256MB memory image and any hard disk chunks modified during that session, the upload data volumes vary significantly due to user activity across the 800+ checkin operations collected. Consequently, we present the data as a cumulative distribution function (CDF) plots. In the ideal case, most upload sizes would be small; therefore, curves that tend to occupy the upper left corner are better. Note that the ALL policy strictly outperforms the IP policy, which in turn, strictly outperforms the Delta policy.

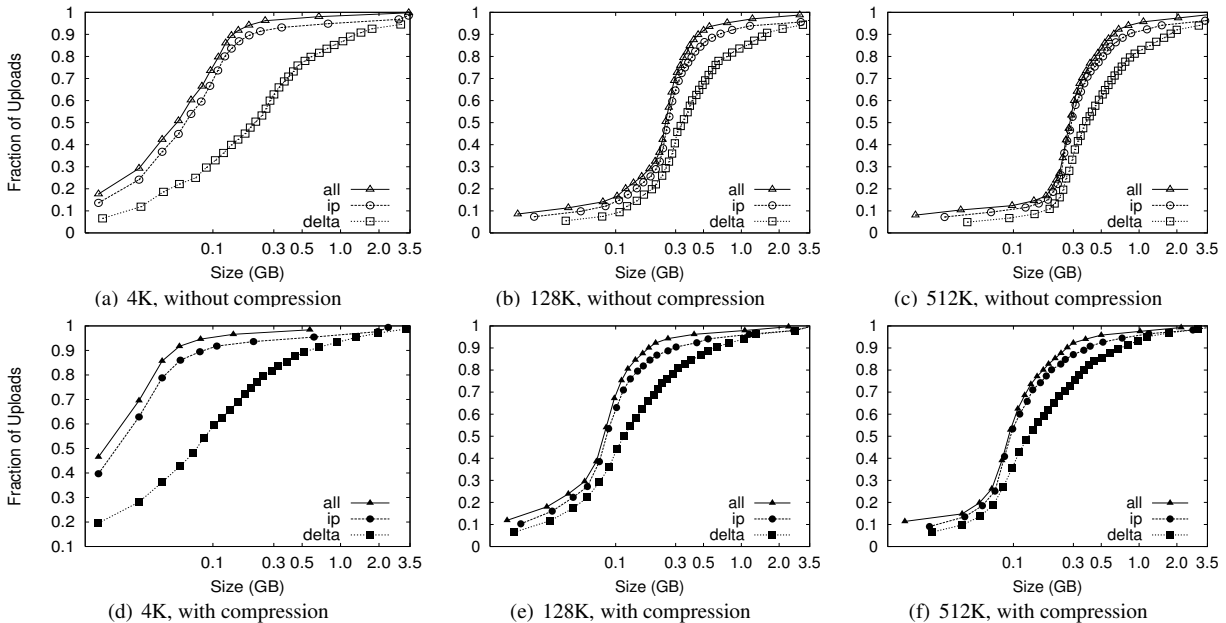


Figure 10: CDF of upload sizes for different policies, without and with the use of compression.

The median (50th percentile) and 95th percentile sizes from Figure 10 are presented along with average upload sizes in Figure 11. Note that the median upload sizes tend to be substantially better than the mean sizes, indicating that the tail of the distribution is somewhat skewed in that the user will see a smaller than average upload sizes for 50% of the upload attempts. Even so, we see from Figure 11(c) that the tail is not so unwieldy as to present sizes more than a factor of 2 to 4 over the average upload size 95% of the time.

Figure 11(a) shows that, for the 128 KB chunksize used in the deployment, the use of CAS reduces the average upload size from 880 MB (Delta policy) to 340 MB (ALL policy). The use of compression reduces the upload size to 293 MB for Delta and 132 MB for ALL. Further, CAS policies provide the most significant benefits where they are needed most, for large upload sizes. From Figure 11(b) we see that CAS improves small upload operations by a modest 20 to 25 percent, while from Figure 11(c), we see that CAS improves the performance of large uploads by a factor of 2 to 5 without compression, and by a factor of 1.5 to 3 with compression. Thus, we observe that CAS significantly reduces the volume of data transmitted during upload operations, and hence the wait time experienced at the end of a user session.

CAS outperforms compression. Figure 11(a) indicates that the ALL policy *without* compression outperforms the Delta policy *with* compression for chunk sizes less than 64 KB (as does the IP policy at a 4 KB chunk size). This shows that for our application, inter-chunk

CAS techniques may identify and eliminate more redundancy than traditional intra-chunk compression techniques. The difference may be substantial, particularly when the upload size is large. As Figure 11(c) shows, the ALL policy *without* compression (chunksize=4 KB) outperforms the Delta policy *with* compression (chunksize=512 KB) by a factor of 4.

IP identifies both temporal and spatial redundancy. For each of the components of Figure 10, we see that the IP policy consistently outperforms the Delta policy. Both of these policies restrict the search space for redundancy identification to a single parcel. However, the Delta policy only detects temporal redundancy between the current and last versions of the parcel, while the IP policy detects temporal and spatial redundancy across all versions of the parcel. The savings of IP over Delta indicate that users often create modified chunks in their environment that either existed at some point in the past, or in another location within the parcel.

ALL identifies inter-parcel savings. In all of Figure 10, the common observation between an IP and ALL comparison is that the ALL policy consistently outperforms the IP policy. This observation is consistent with our intuition that for upload operations, the ALL policy must perform *at least* as well as the IP policy because the ALL policy identifies redundancy within the set of blocks visible to the IP policy as well as blocks in other parcels. In fact, Figure 11(a) indicates that the ALL policy performs about twice as well as the IP policy for small chunk sizes and approximately 25 percent better

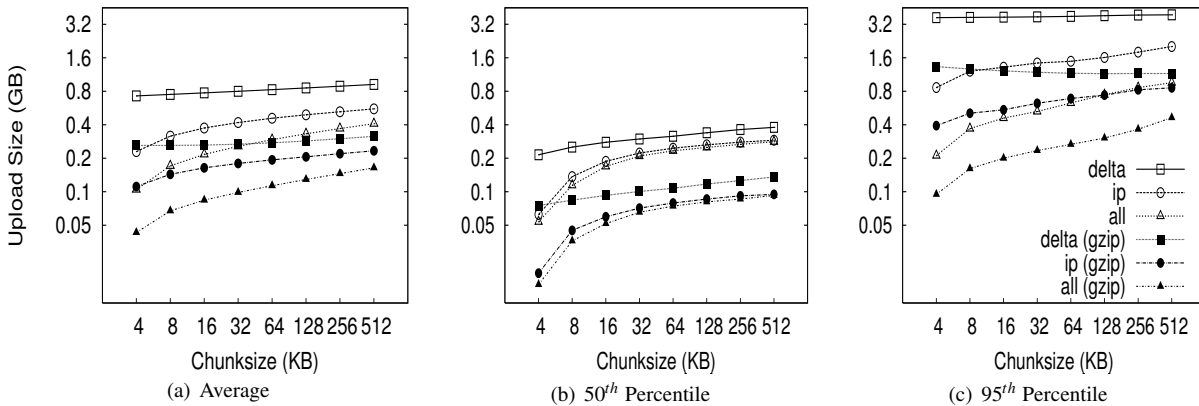


Figure 11: Upload sizes for different chunk sizes.

at larger chunk sizes.

This difference shows the benefit of having a larger pool of candidate chunks when searching for redundant data. As mentioned, one source of this gain can be the “broadcast” of objects to many users (e.g. from software installation, patches, popular documents, big email attachments, etc.). In systems leveraging the ALL policy, therefore, operations that might be expected to impose a significant burden such as the distribution of security patches may result in very little realized cost because the new data need only be stored once and transmitted once (across all users in the system).

5.2 Effect of Chunksize on Networking

The choice of chunksize will affect both the download size and upload size to a server. We continue our discussion of upload operations first, and then discuss the appropriate chunksize for download operations.

5.2.1 Effect on Upload Size

Smaller chunksize is better for CAS. Figure 11(a) shows very clearly that smaller chunk sizes result in more efficient upload transmission for CAS policies. In fact, under the ALL policy, users with 4 KB chunk sizes will experience average upload sizes that are approximately one-half the average size experienced by users with a 128 KB chunk size (whether compression is employed or not).

Chunk sizes of 4 KB turned out to be optimal for all policies when considering the average upload size. However, chunksize plays a very limited role for the non-CAS (Delta) policy, and Figure 11(c) indicates that smaller chunk sizes may even be a liability for transfer size outliers under the Delta policy with compression.

5.2.2 Effect on Download Size

Employing CAS techniques also potentially affects the volume of data transmitted during download operations in two ways. First, CAS can identify intra-version redundancy and reduce the total volume of data transmission. Second, when a user requests a download of their environment to a particular client, CAS has the potential to expose any chunks selected for download that are identical to chunks that happen to have been cached on that client from previous sessions.

To simplify our discussion we assume that the client has cached at most one previous version of the parcel in question, and if a cached version is present, it is the version prior to the one requested for download. This assumption corresponds to an expected common user telecommuting behavior. Namely, the user creates version $N - 1$ of a parcel at home and uploads it to the server. The user then retrieves version $N - 1$ at work, creates version N , and uploads that to the server. Our operation of interest is the user’s next download operation at home; upon returning home, the user desires to download version N and modify it. Fortunately, the user may still have version $N - 1$ cached locally, and thus, only the modified data that does not exist in the cache need be retrieved. Note that this CAS technique can be likened to a sub-set of the IP policy which inspects chunks of a single user, but only for a single previous version.

Our client management system, ISR, supports two basic modes for download: *demand-fetch* and *complete-fetch*. Demand-fetch mode instantiates the user’s environment after downloading the minimum data needed to reconstruct the user’s environment, essentially the physical memory image corresponding to the user’s VM (256 MB in our test deployment). In particular, the largest portion of the VM image, the virtual disk drive, is *not* retrieved before instantiating the user’s environment.

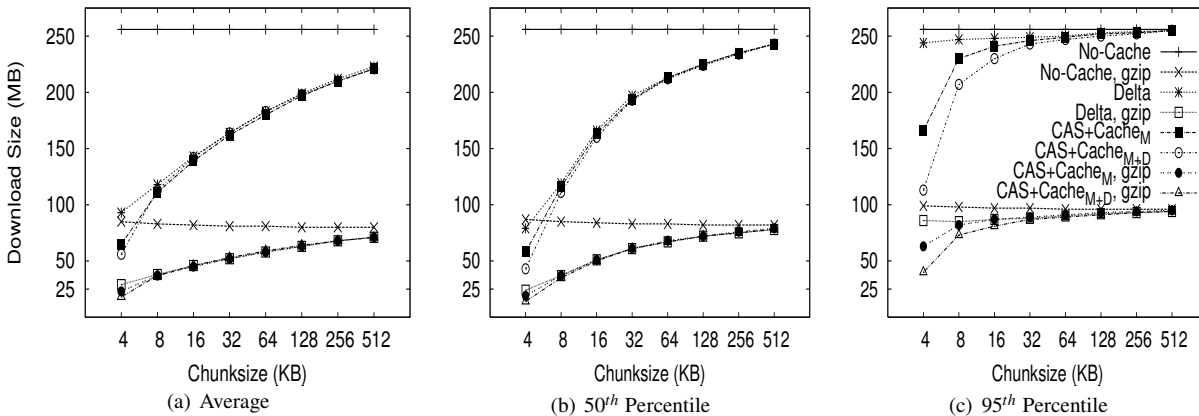


Figure 12: Download size when fetching memory image of latest version.

During operation, missing data blocks (chunks) must be fetched on demand in a manner analogous to demand-paging in a virtual memory system. The complete-fetch mode, in contrast, requires that the entire VM image including the virtual disk image (8.25 GB in our test deployment) be present at the client before the environment is instantiated.

Caching improves demand-fetch. To evaluate the effect of client-side caching on demand-fetch download volume, we calculated how much data would need to be transferred from the server to a client under various conditions and collected those results in Figure 12. The curve labeled “No-cache” depicts the volume of data that would be transmitted if no data from the previous version of the parcel were present in the client cache. Under the “Delta” policy, the chunks in the memory image are compared with the same chunks (those at the same offset within the image) in the previous version of the memory image to determine whether they match. The “CAS+Cache_M” policy compares the keyring for the new memory image with the keyring for the previous memory image to determine which chunks need to be transferred. The “CAS+Cache_{M+D}” policy is similar except that it searches all the data cached on the client (memory *and* disk) to identify chunks that are already present on the client. Each basic curve in Figure 12 also has a companion curve depicting the download volumes observed when compression is employed during the transfer.

As shown in Figure 12(a), introducing a differencing mechanism (either Delta or CAS) yields a reduction of approximately 20% (for the 128 KB chunk size) in the download size relative to the size when no cached copy is present. Using compression alone, however, is very effective—reducing the transfer size from 256 MB to approximately 75 MB in the absence of caching. Leveraging cached data in addition to compression yields a

further 20% reduction.

Chunk size dramatically affects demand-fetch.

Moving to a smaller chunk size can have a significant effect on the volume of data transmitted during a download operation, particularly if compression is not used, as shown in Figure 12. The average download size, in particular, is reduced by a factor of two (for Delta) to four (for “CAS+Cache_{M+D}”) when the chunk size is reduced from 128 KB to 4 KB when comparing the policies either with or without compression. Further, we see again that, with a 4 KB chunk size, the CAS policies *without* compression outperform the no-cache policy *with* compression.

The difference between the “CAS+Cache_M” and “CAS+Cache_{M+D}” policies is also most apparent with a 4 KB chunk size. At this size, in the absence of compression, leveraging the cached disk image in addition to the memory image reduces the average transfer size to 56 MB from the 65 MB required when leveraging just the memory image. A similar gain is observed when compression is employed; the transfer size is reduced from 23 MB (for “M”) to 18 MB (for “M+D”)—a savings of more than 20%.

However, the added benefit of inspecting additional cached data diminishes quickly as the chunk size increases beyond 4 KB. We believe this phenomenon is due, at least in part, to the fact that the 4 KB size corresponds to the size of both memory pages disk blocks in these VMs. Consequently, potentially redundant data is most likely to be exposed when chunks are aligned to 4 KB boundaries.

Caching significantly improves complete-fetch. The need for efficient download mechanisms is perhaps greatest in the complete-fetch mode due to the volume of data in question. In this mode, the user is requesting the download of the entire VM image, the most signifi-

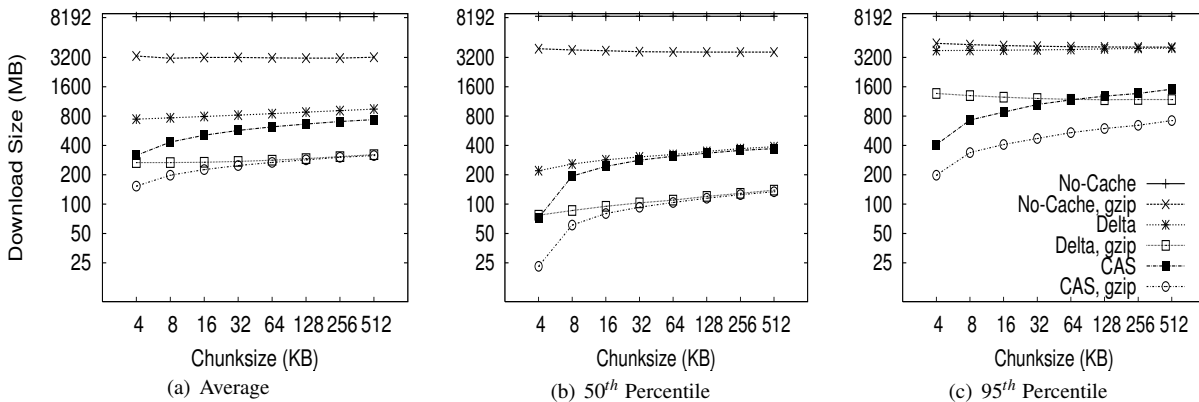


Figure 13: Download size when fetching memory and disk of latest version.

cant component of which is the virtual disk drive image. In our test deployment, the virtual disk drive was a very modest 8 GB in size. One can readily imagine that users might desire virtual disk drive spaces an order of magnitude larger. However, even with a modest size (8 GB) and a fast network (100 Mbps), a complete-fetch download will require at least 10 minutes. Consequently, reducing the volume of data to be transferred by at least an order of magnitude is essential to the operation of these client management systems.

The basic tools are the same as those mentioned for demand-fetch mode. That is, a cache of at least one previous version of the parcel is maintained at the client, if possible. Redundancy between the cached version and the current version on the server is identified and only non-redundant chunks are transferred during the download. Further, the transferred chunks are (optionally) compressed prior to transmission. One difference between our treatment of demand-fetch and complete-fetch is that the CAS policy for complete-fetch mode always compares the entire current server version with the entire cached client version. Consequently, Figure 13 includes a single “CAS” curve rather than the separate “M” and “M+D” curves of Figure 12.

Figure 13(a) indicates that intelligent transfer mechanisms can, in fact, significantly reduce the volume of data transmitted during a complete-fetch operation. Compression reduces the average data volume from 8394 MB to 3310 MB, a factor of 2.7. In contrast, the Delta policy *without* compression yields a factor of 9.5 and a factor of 28.6 *with* compression, assuming a 128 KB chunk size. At the same chunk size, CAS provides even more impressive savings: factors of 12.6 and 29.5, without and with compression, respectively.

Small chunk sizes yield additional savings. While the slopes of the “CAS” and “CAS,gzip” curves are not as dramatic as in previous figures, reducing the chunk

size from 128 KB to 4 KB still yields significant savings. At this chunk size, the average download size shrinks from the nominal 8+ GB size by a factor of 31.4 without compression and a factor of 55 (*fifty-five!*) by employing both CAS and compression.

CAS has a big impact where it’s needed most. Figure 13(c) indicates that the 4 KB “CAS,gzip” combination may be particularly effective for download operations that may otherwise have resulted in large data transfers. The performance gap between “CAS,gzip” and “Delta,gzip” is particularly large in this graph. In fact, for small chunk sizes “CAS” *without* compression significantly outperforms the Delta policy *with* compression. Note in particular that when employing the “CAS,gzip” policy with the 4 KB chunk size, the 95th percentile upload sizes are not significantly larger than the average size, thus providing the user with better expected bounds on the time required for a complete-fetch download.

6 Related Work

Our results are most directly applicable to VM-based client management systems such as the Collective [7, 35], Soulpad [5], and ISR [19, 37], as well as systems that use VMs for Grid applications [9, 14, 22, 24, 39]. Further, our results also provides guidelines for the storage design of applications that need to version VM history. Examples include intrusion detection [12], operating systems development [18], and debugging system configurations [46]. Related applications include storage cluster and web services where VMs are being used for balancing load, increasing availability, and simplifying administration [28, 45].

The study could also help a large number of systems that use CAS to improve storage and network utilization. Examples of CAS-based storage systems include EMC’s Centera [13], Deep Store [48], the Venti [30], the

Pastiche [10] backup system, the TAPER [17] scheme for replica synchronization and Farsite [2]. Other systems use similar CAS-based techniques to eliminate duplicate data at various levels in the network stack. Systems such as the CASPER [42] and LBFS [27] file systems, Rhea et al.'s CAS-enabled WWW [33], etc. apply these optimizations at the application layer. Other solutions such as the DOT transfer service [41] and Riverbed's WAN accelerator [34] use techniques such as Rabin Fingerprinting [26, 31, 38] to detect data duplication at the transfer layer. However, most of these systems have only concentrated on the mechanism behind using CAS. Apart from Bolosky et al. [3] and Policroniades and Pratt [29], there have been few studies that measure data commonality in real workloads. The study in this paper helps by providing a point of reference for commonality seen in VM migration workloads.

7 Conclusions

Managing large volumes of data is one of the major challenges inherent in developing and maintaining enterprise client management systems based on virtual machines. Using empirical data collected during seven-months of a live-deployment of one such system, we conclude that leveraging content addressable storage (CAS) technology can significantly reduce the storage and networking resources required by such a system (questions Q1 and Q2 from Section 1).

Our analysis indicates that CAS-based management policies typically benefit from dividing the data into very small chunk sizes despite the associated meta-data overhead. In the absence of compression, 4 KB chunks yielded the most efficient use of both storage and network resources. At this chunk size, a privacy-preserving CAS policy can reduce the system storage requirements by approximately 60% when compared to a block-based differencing policy (*Delta*), and a savings of approximately 80% is possible by relaxing privacy.

Similarly, CAS policies that leverage data cached on client machines reduce the average quantity of data that must be transmitted during both upload and download operations. For upload, this technique again results in a savings (compared to *Delta*) of approximately 70% when preserving privacy and 80% when not. This technique also reduces the cost of *complete-fetch* download operations by more than 50% relative to the *Delta* policy (irrespective of CAS privacy policy) and by more than an order of magnitude relative to the cost when caching is not employed.

Leveraging compression in addition to CAS techniques provides additional resource savings, and the combination yields the highest efficiency in all cases. However, a surprising finding from this work is that CAS alone yields higher efficiency for this data set than com-

pression alone, which is significant because the use of compression incurs a non-zero runtime cost for these systems.

Acknowledgments

This work is sponsored in part by NSF under grants CNS-0509004 and IIS-0429334, in part by a subcontract from SCEC as part of NSF ITR EAR-01-22464, and in part by support from Intel and CMU CyLab. Our heartfelt thanks to Intel's Casey Helfrich for his assistance developing the software and deployment, Dave Bantz and Ramón Cáceres at IBM for the donation of the BladeCenter, Mark Poepping and his team at CMU computing services for housing the server, and Bob Cosgrove, Tod Pike, and Mark Puskar in the CMU School of Computer Science facilities group for helping integrate our experiment into the campus environment. All unidentified trademarks are the property of their respective owners.

References

- [1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.
- [2] W. J. Bolosky, S. Corbin, D. Goebel, , and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, 2000.
- [3] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. *ACM SIGMETRICS Performance Evaluation Review*, 28(1):34–43, 2000.
- [4] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.
- [5] R. Cáceres, C. Carter, C. Narayanaswami, and M. Raghunath. Reincarnating PCs with portable SoulPads. In *MobiSys '05: Proceedings of the 3rd International conference on Mobile Systems, Applications, and Services*, 2005.
- [6] B. Calder, A. A. Chien, J. Wang, and D. Yang. The entropia virtual machine for desktop grids. In *VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution environments*, 2005.
- [7] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A Cache-Based System Management Architecture. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.
- [8] P. Chen and B. Noble. When Virtual is Better Than Real. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems*, 2001.
- [9] S. Childs, B. A. Coghlan, D. O'Callaghan, G. Quigley, and J. Walsh. A single-computer grid gateway using virtual machines. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications (AINA 2005)*, pages 310–315, Taipei, Taiwan, Mar. 2005.
- [10] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. In *OSDI: Symposium on Operating Systems Design and Implementation*, 2002.
- [11] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS '02: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 617, Washington, DC, USA, 2002. IEEE Computer Society.

- [12] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *OSDI*, 2002.
- [13] EMC Corporation. *EMC Centera Content Addressed Storage System*, 2003. <http://www.emc.com/>.
- [14] R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*, page 550, Washington, DC, USA, 2003. IEEE Computer Society.
- [15] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.
- [16] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource Management using Virtual Clusters on Shared-Memory Multiprocessors. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, 1999.
- [17] N. Jain, M. Dahlin, and R. Tewari. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *USENIX Conference on File and Storage Technologies*, Dec 2005.
- [18] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *Proceedings of the USENIX 2005 Annual Technical Conference*, pages 1–15, Anaheim, CA, Apr. 2005.
- [19] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Fourth IEEE Workshop on Mobile Computing Systems and Applications*, Callicoon, New York, June 2002.
- [20] M. A. Kozuch, C. J. Helfrich, D. O'Hallaron, and M. Satyanarayanan. Enterprise Client Management with Internet Suspend/Resume. *Intel Technology Journal*, November 2004.
- [21] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.
- [22] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 7, Washington, DC, USA, 2004. IEEE Computer Society.
- [23] P. Kulkarni, F. Douglass, J. D. LaVoie, and J. M. Tracey. Redundancy Elimination Within Large Collections of Files. In *USENIX Annual Technical Conference, General Track*, 2004.
- [24] B. Lin and P. Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC 2005)*, Seattle, WA, Nov. 2005.
- [25] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, 2004.
- [26] U. Manber. Finding Similar Files in a Large File System. In *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.
- [27] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.
- [28] M. Nelson, B.-H. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the USENIX 2005 Annual Technical Conference*, Anaheim, CA, Apr. 2005.
- [29] C. Policoniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *USENIX Annual Technical Conference, General Track*, 2004.
- [30] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, 2002.
- [31] M. Rabin. Fingerprinting by Random Polynomials. In *Harvard University Center for Research in Computing Technology Technical Report TR-15-81*, 1981.
- [32] J. Reumann, A. Mehra, K. G. Shin, and D. D. Kandlur. Virtual Services: A New Abstraction for Server Consolidation. In *USENIX Annual Technical Conference, General Track*, 2000.
- [33] S. Rhea, K. Liang, and E. Brewer. Value-Based Web Caching. In *Proceedings of the Twelfth International World Wide Web Conference*, 2003.
- [34] Riverbed Technology, Inc. <http://www.riverbed.com>.
- [35] C. Sapuntzakis and M. S. Lam. Virtual Appliances in the Collective: A Road to Hassle-Free Computing. In *Proceedings of the Ninth Workshop on Hot Topics in Operating System*, May 2003.
- [36] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.
- [37] M. Satyanarayanan, M. A. Kozuch, C. J. Helfrich, and D. R. O'Hallaron. Towards Seamless Mobility on Pervasive Hardware. *Pervasive and Mobile Computing*, 1(2):157–189, July 2005.
- [38] N. T. Spring and D. Wetherall. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of ACM SIGCOMM*, August 2000.
- [39] A. I. Sundararaj and P. A. Dinda. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 177–190, San Jose, CA, May 2004.
- [40] N. Taesombut and A. Chien. Distributed Virtual Computer (DVC): Simplifying the Development of High Performance Grid Applications. In *Proceedings of the Workshop on Grids and Advanced Networks (GAN'04)*, 2004.
- [41] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for internet data transfer. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, May 2006.
- [42] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.
- [43] M. Vilayannur, P. Nath, and A. Sivasubramaniam. Providing Tunable Consistency for a Parallel File Store. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST'05)*, 2005.
- [44] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Boston, MA, USA, 2002.
- [45] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing Storage for a Million Machines. In *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS)*, 2005.
- [46] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *OSDI*, 2004.
- [47] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Dec 2002.
- [48] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE '05)*, April 2005.
- [49] Y. Zhang, A. Bestavros, M. Guirguis, I. Matta, and R. West. Friendly Virtual Machines: Leveraging a Feedback-Control Model for Application Adaptation. In *VEE '05: Proc. 1st ACM/USENIX Inter. Conf. on Virtual Execution Envs*, 2005.
- [50] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Compare-by-Hash: A Reasoned Analysis

J. BLACK *

April 14, 2006

Abstract

Compare-by-hash is the now-common practice used by systems designers who assume that when the digest of a cryptographic hash function is equal on two distinct files, then those files are identical. This approach has been used in both real projects and in research efforts (for example `rsync` [16] and LBFS [12]). A recent paper by Henson criticized this practice [8]. The present paper revisits the topic from an advocate's standpoint: we claim that compare-by-hash is completely reasonable, and we offer various arguments in support of this viewpoint in addition to addressing concerns raised by Henson.

Keywords: Cryptographic hash functions, distributed file systems, probability theory

1 Introduction

There is a well-known short-cut technique for testing two files for equality. The technique entails using a cryptographic hash function, such as SHA1 or RIPEMD-160, to compare the files: instead of comparing them byte-by-byte, we instead compare their hashes. If the hashes differ, then the files are certainly different; if the hashes agree, then the files are almost certainly the same. The motivation here is that the two files in question might live on opposite ends of a low-bandwidth network connection and therefore a substantial performance gain can be realized by sending a small (eg, 20 byte) hash digest rather than a large (eg, 20 megabyte) file. Even without this slow network connection, it is still a significant performance gain to compare hashes rather than compare files directly when the files live on the same disk: if

we always keep the latest hash value for files of interest, we can quickly check the files for equality by once again comparing only a few bytes rather than reading through them byte-by-byte.

One well-known use of this technique is the file synchronizing tool called `rsync` [16]. This tool allows one to maintain two identical copies of a set of files on two separate machines. When updates are applied to some subset of the files on one machine, `rsync` copies those updates to the other machine. In order to determine which files have changed and which have not, `rsync` compares the hashes of respective files and if they match, assumes they have not changed since the last synchronization was performed.¹

Another well-known use of compare-by-hash in the research community is the Low-Bandwidth File System (LBFS) [12]. The LBFS provides a fully-synchronized file system over low-bandwidth connections, once again using a cryptographic hash function (this time SHA1) to aid in determining when and where updates have to be distributed.

Cryptographic hash functions are found throughout cryptographic protocols as well: virtually every digital signature scheme requires that the message to be signed is first processed by a hash function, for example. Time-stamping mechanisms, some message-authentication protocols, and several widely-used public-key cryptosystems also use cryptographic hash functions.

A difference between the above scenarios is that usually in the compare-by-hash case (`rsync` and LBFS are our examples here) **there is no adversary**. Or at least the goal of the designers was not to provide security via the use of cryptographic hash functions. So in some sense using a cryptographic hash function is overkill: a hash function that is

* Department of Computer Science, 430 UCB, Boulder, Colorado 80309 USA. E-mail: jrblack@cs.colorado.edu WWW: www.cs.colorado.edu/~jrblack/

¹This is an oversimplification; `rsync` actually uses a lightweight “rolling” hash function in concert with a cryptographic hash function, MD4, on blocks of the files being compared. The simplified description will suffice for our purposes.

simply “good” at spreading distinct inputs randomly over some range should suffice. In the cryptographic examples, there *is* an adversary: indeed all the latter examples are taken from the cryptographic literature and the hash functions in these scenarios are specifically introduced in order to foil a would-be attacker.

In spite of the fact these functions are stronger than what is needed, a recent paper by Henson [8] criticizes the approach and gives a list of concerns. In this paper we will revisit some of the issues raised by Henson and argue that in most cases they are not of great concern. We will argue in favor of continuing to use compare-by-hash as a cheap and practical tool.

2 The Basics of Hash Functions

They are variously called “cryptographic hash functions,” “collision-resistant hash functions,” and “Universally One-Way Hash Functions,” not to mention the various acronyms in common usage. Here we will just say “hash function” and assume we mean the cryptographic sort.

A hash function h accepts any string from $\{0,1\}^*$ and outputs some b -bit string called the “hash” or “digest.” The most well-known hash functions are MD4, MD5 [14], RIPEMD-160 [6], and SHA1 [7]. The first two functions output digests of 128 bits, and the latter two output 160 bits.

Although it is not possible to rigorously define the security of these hash functions, we use the following (informal) definitions to capture the main goals.² A hash function h should be

- **Collision-Resistant:** it should be “computationally intractable” to find distinct inputs a and b such that $h(a) = h(b)$. Of course such a and b *must* exist given the infinite domain and finite range of h , but finding such a pair should be very hard.
- **Inversion-Resistant:** given any digest v , it should be “computationally intractable” to find an input a such that $h(a) = v$. This means that hashing a document should provide a “fingerprint” of that document without revealing anything about it.
- **Second-Preimage-Resistant:** given an input a and its digest $v = h(a)$, it should be “computationally intractable” to find a second input

²For a proper discussion of these notions, along with a discussion of how they relate to one another, see [15].

$b \neq a$ with $h(b) = v$. This is the condition necessary for secure digital signatures, which is the context in which hashing is most commonly employed cryptographically.

In the context of compare-by-hash, collision-resistance is our main concern: if ever two distinct inputs produced the same digest, we would wrongly conclude that distinct objects were the same and this would result in an error whose severity is determined by the application; typically a file would be out-of-sync or a file system would become inconsistent.

FINDING COLLISIONS. If avoiding a collision (the event that two distinct inputs hash to the same output) is our main goal, we must determine how hard it is to find one. For a random function, this is given by the well-known “birthday bound.” Roughly stated, if we are given a list of b -bit independent random strings, we expect to start seeing collisions after about $2^{b/2}$ strings have appeared. This means that MD4 and MD5 should begin showing collisions after we have seen about 2^{64} hash values, and RIPEMD-160 and SHA1 should begin showing collisions after about 2^{80} hash evaluations.

This is the expected number of hash evaluations before *any* pair of files collide. The probability that a *given* pair of files collide is much lower: 2^{-160} for SHA1, for example.

3 Compare-by-Hash: A Flawed Technique?

In her paper, Henson raises a number of issues which she deems important shortcomings of the compare-by-hash technique [8]. A central theme in Henson’s paper is her opposition to the notion that digests can be treated as unique ids for blocks of data. She gives a list of arguments, often supported by numerical calculations, as to why this is a dangerous notion. In order to take an opposing viewpoint, we now visit several of her most prominent claims and examine them.

INCORRECT ASSUMPTIONS. One of Henson’s most damning claims is her repeated assertion (cf. Sections 3 and 4.1) that in order for the outputs of a cryptographic hash function to be uniform and random, the

inputs to the function must be random.³ In fact, in Section 4.1, she claims that this supposedly-wrong assumption on the inputs is “the key insight into the weakness of compare-by-hash.” She correctly points out that most file-system data are in fact *not* uniform over any domain. However, her assertion about hash functions is incorrect.

Cryptographic hash functions are designed to model “random functions.” A random function from the set of all strings to the set of b -bit values can be thought of as an infinitely-long table where every possible string is listed in the left column, and for each row a random and independent b -bit number is listed in the right column. Obviously with this (fantasy) mapping, even highly-correlated distinct inputs will map to independent and random outputs. Of course any actual hash function we write down cannot have this property (since it is a static object the moment we write it down), most researchers agree that our best hash functions, like SHA1 and RIPEMD-160, do a very good job at mapping correlated inputs to uncorrelated outputs. (This is what’s known as withstanding “differential cryptanalysis.”)

If the assumption that inputs must be random is indeed the key insight into the weakness of compare-by-hash, as Henson claims, then we argue that perhaps compare-by-hash is not so weak after all.

ESTABLISHED USES OF HASHING. In section 3, Henson states that “compare-by-hash” sets a new precedent by relying solely on hash-value comparison in place of a direct comparison. While new precedents are not necessarily a bad thing, in this case it is simply untrue: one of the oldest uses of cryptographic hashing is (unsurprisingly) in cryptography. More specifically, they are universally used in digital signature schemes where a document being digitally signed is first hashed with a cryptographic hash function, and then the signature is applied to the hash value. This is expressly done for performance reasons: applying a computationally-expensive digital signature to a large file would be prohibitive, but applying it to a short hash-function output is quite practical. The unscrupulous attacker now need only find a different (evil) file with the same hash value and this signature will appear valid for it as well. Therefore the security of the scheme rests squarely on the strength of the hash function used (in addition to the strength of

the signature scheme itself), and the comparison of files is never performed: it is done entirely through the hash function. In fact, this could fairly be called “compare-by-hash” as well, and it has been accepted by the security community for decades.

QUESTIONABLE EXAMPLES. Henson notes that if a massive collision-finding attempt for SHA1 were mounted by employing a large distributed brute-force attack using a compare-by-hash-based file system that also uses SHA1, collisions would not be detected. First, the example is a bit contrived: using a file-system based on SHA1 to look for collisions in SHA1 is a bit like testing NFS by mirroring it with the exact same implementation of NFS and then comparing to see if the results match. But even given this, it’s still not clear that searching for SHA1 collisions using a SHA1-based compare-by-hash file system would present any problems. Van Oorschot and Wiener have described the best-known ways of doing parallel collision searching in this domain [17]. Let’s use SHA1 in an example: each computer selects a (distinct, random) starting point x_0 , and then iterates the hash function $x_i = \text{SHA1}(x_{i-1})$, looking for “landmark” values that are stored in the file system. (These landmark values might be hash values with 40 leading zeros, for example. Under the assumption the hash outputs are uniform and random, we would expect to see such a point every 2^{40} iterations. We do this in order to avoid storing *all* hash values which for SHA1 would approximately require an expected 2^{85} bytes of storage, not including overhead for data structures for collision detection!) Since these landmark values would be written to the file system (along with other bookkeeping information all stored in some data structure), and since the number of blocks would be *far* less than 2^{80} , it’s highly unlikely that even a SHA1-based compare-by-hash file system would have any difficulties at all.

The second example Henson gives is the VAL-1 hash function. The VAL-1 hash creates a publicly-known collision on two points, zero and one, and otherwise behaves like SHA1. Henson claims the VAL-1 hash has “almost identical probability of collision” as SHA1 and yet allows a user to make changes that would go undetected in a compare-by-hash-based file system. (The term “collision probability” is not defined anywhere in her paper.) Of course it is correct that the probability of collision is nearly the same between VAL-1 and SHA1 when the inputs are random (which itself is difficult to define as we mentioned

³Technically, this statement cannot make sense since the input set is infinite and not compact and therefore cannot even have a probability measure.

above). But more to the point, VAL-1 does not have *security* equivalent to SHA1 by any normal notion of hash-function security in common use. The informal notion of collision-resistance given above dictates that it is intractable to find collisions in our hash function; in VAL-1 it is quite trivial since $\text{VAL-1}(0) = \text{VAL-1}(1)$. (In the formal definitions for hash-function security, VAL-1 would fail to be secure as well.) Once again, the example is dubious due to the assumption that hash-function security rests solely on the distribution of digests over randomly-chosen inputs.

WHAT IS THE ATTACK MODEL? Although Henson mentions correctness as the principal concern, she also mentions security. But throughout the paper it's unclear what the attack model is, exactly. More specifically, if we are worried about collisions, do we assume that our enemy is just bad luck, or is there an intelligent attacker trying to cause collisions?

If we're just concerned with bad luck, we have seen there is very little to worry about: the chance that two files will have the same hash (assuming once again that the hash function adequately approximates a random function) is about 2^{-160} for a 160-bit hash function like SHA1 or RIPEMD-160. If there is an active attacker *trying* to cause trouble, he must first somehow find a collision. This is a difficult proposition, but in the extremely unlikely case he succeeds, he may find blocks b_1 and b_2 that collide. In this case, he may freely cause problems by substituting b_1 for b_2 or vice-versa, and a compare-by-hash file system will not notice. However, finding this one collision (which we must emphasize has still not been done to-date) would not enable him to alter arbitrary blocks. If the method by which he found b_1 and b_2 was to try random blocks until he found a collision, it's highly unlikely that b_1 or b_2 are blocks of interest to him, that they have any real meaning, or that they even *exist* on the file system in question.

In either case, it would seem that compare-by-hash holds up well in both attack models.

THE INSECURITY OF CRYPTOGRAPHIC OBJECTS. Henson spends a lot of time talking about the poor track-record of cryptographic algorithms (cf. Section 4.2), stating that the literature is "rife with examples of cryptosystems that turned out not to be nearly as secure as we thought." She further asserts that "history tells us that we should expect any popular cryptographic hash to be broken within a few years

of its introduction." Although it is true that some algorithms are broken, this is by no means as routine as she implies.

The Davies-Meyers hash has been known since 1979, and the Matyas-Meyer-Oseas scheme since 1985 [11]. These have never been broken despite their being very well-known and well-analyzed [13, 2]. RIPEMD-160 and SHA1 were both published more recently (1995) but have still held up for more than just "a few years," since there is still no known practical attack against either algorithm. While it is true that there have been many published cryptographic ideas which were later broken, rarely have these schemes ever made it into FIPS, ANSI, or ISO standards. The vetting process usually weeds them out first. Probably the best example is the DES blockcipher which endured 25 years of analysis without any effective attacks being found.

Even when these "breaks" occur, they are often only of theoretical interest: they break because they fail to meet some very strict definition of security set forth by the cryptographic community, not because the attack could be used in any practical way by a malicious party.

Ask any security expert and he or she will tell you the same thing: if you want to subvert the security of a computer system, breaking the cryptography is almost never the expeditious route. It's highly more likely that there is some flaw in the system itself that is easier to discover and exploit than trying to find collisions in SHA1.

That said, there *have* been attacks on cryptographic hash functions over the past 10 years, and some very striking ones just in just the past few years. We briefly discuss these.

RECENT ATTACKS ON CRYPTOGRAPHIC HASH FUNCTIONS. Collisions in MD4 were found by Hans Dobbertin in 1996 [5]. However, MD4 is still used in `rsync` undoubtedly due to the arguments made above: there is typically no adversary, and just *happening* on a collision in MD4 is highly unlikely. MD4 was known to have shortcomings long before Dobbertin's attack, so its inventor (Ron Rivest) also produced the stronger hash function MD5. MD5 was also attacked by Dobbertin, with partial success, in 1996 [4]. However, full collisions in MD5 were not found until 2004 when Xiaoyun Wang shocked the community by announcing that she could find collisions in MD5 in a few hours on an IBM P690 [19]. Since that time, other researchers have refined her

attack to produce collisions in MD5 in just a few minutes on a Pentium 4, on average [9, 1]. Clever application of this result has allowed various attacks: distinct X.509 certificates with the same MD5 digest [10], distinct postscript files with the same MD5 digest [3], and distinct Linux binaries with the same MD5 digest [1].

No inversion or second-preimage attacks have yet been found, but nonetheless cryptographers now consider MD5 to be compromised. SHA1 and RIPEMD-160 still have no published collisions, but many believe it is only a matter of time. Wang's latest attacks claim an attack complexity of around 2^{63} , well within the scope of a parallelized attack [18]. However, until the attack is implemented, we won't know for sure if her analysis is accurate.

Given all of this, we can now revisit the central question of this paper one final time: "is it safe to use cryptographic hash functions for compare-by-hash?" I argue that it is: even with MD4 (collisions can be found by hand in MD4!). This is, once again, because in the typical compare-by-hash setting there is no adversary. The event that two arbitrary distinct files will have the same MD4 hash is highly unlikely. And in the `rsync` setting, comparisons are done between files only if they have the same filename; we don't compare all possible pairs of files across the filesystems. No birthday phenomenon exists here.

In the cryptographic arena, the issue is more complicated: there *is* an adversary and what can be done with the current attack technology is a topic currently generating much discussion. Where this ends up remains to be seen.

4 Conclusion

We conclude that it is certainly fine to use a 160-bit hash function like SHA1 or RIPEMD-160 with compare-by-hash. The chances of an accidental collision is about 2^{-160} . This is unimaginably small. You are roughly 2^{90} times more likely to win a U.S. state lottery *and* be struck by lightning simultaneously than you are to encounter this type of error in your file system.

In the adversarial model, the probability is higher but consider the following: it was estimated that general techniques and custom hardware could find collisions in a 128-bit hash function for 10,000,000 USD in an expected 24 days [17]. Given that this estimate was made in 1994 we could use Moore's law to

extrapolate the cost in 2006 to be more like 80,000 USD. Since SHA1 has 32 more bits, we could rescale these estimates to be 80,000,000 USD and 2 years to find a collision in SHA1. This cost is far out of reach for anyone other than large corporations and governments. But if we are worried about adversarial users with lots of resources, it might make sense to use the new SHA-256 instead.

Of course, if someone is willing to spend 80,000,000 USD to break into your computer system, it would probably be better spent finding vulnerabilities in the operating system or on social engineering. And this approach would likely yield results in a lot less than 2 years.

References

- [1] BLACK, J., COCHRAN, M., AND HIGHLAND, T. A study of the MD5 attacks: Insights and improvements. In *Fast Software Encryption* (2006), Lecture Notes in Computer Science, Springer-Verlag.
- [2] BLACK, J., ROGAWAY, P., AND SHRIMPTON, T. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In *Advances in Cryptology – CRYPTO '02* (2002), vol. 2442 of *Lecture Notes in Computer Science*, Springer-Verlag.
- [3] DAUM, M., AND LUCKS, S. Attacking hash functions by poisoned messages. Presented at the rump session of Eurocrypt '05.
- [4] DOBBERTIN, H. Cryptanalysis of MD5 compress. Presented at the rump session of Eurocrypt '96.
- [5] DOBBERTIN, H. Cryptanalysis of MD4. In *Fast Software Encryption* (1996), vol. 1039 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 53–69.
- [6] DOBBERTIN, H., BOSSELAERS, A., AND PRENEEL, B. Ripemd-160, a strengthened version of ripemd. In *3rd Workshop on Fast Software Encryption* (1996), vol. 1039 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 71–82.
- [7] FIPS 180-1. Secure hash standard. NIST, US Dept. of Commerce, 1995.

- [8] HENSON, V. An analysis of compare-by-hash. In *HotOS: Hot Topics in Operating Systems* (2003), USENIX, pp. 13–18.
- [9] KLIMA, V. Finding MD5 collisions on a notebook PC using multi-message modifications. In *International Scientific Conference Security and Protection of Information* (May 2005).
- [10] LENSTRA, A. K., AND DE WEGER, B. On the possibility of constructing meaningful hash collisions for public keys. In *Information Security and Privacy, 10th Australasian Conference – ACISP 2005* (2005), vol. 3574 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 267–279.
- [11] MATYAS, S., MEYER, C., AND OSEAS, J. Generating strong one-way functions with cryptographic algorithms. *IBM Technical Disclosure Bulletin* 27, 10a (1985), 5658–5659.
- [12] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *SOSP: ACM Symposium on Operating System Principles* (2001), pp. 174–187.
- [13] PRENEEL, B., GOVAERTS, R., AND VANDEWALLE, J. Hash functions based on block ciphers: A synthetic approach. In *Advances in Cryptology – CRYPTO ’93* (1994), Lecture Notes in Computer Science, Springer-Verlag, pp. 368–378.
- [14] RFC 1321. The MD5 message digest algorithm. IETF RFC-1321, R. Rivest, 1992.
- [15] ROGAWAY, P., AND SHRIMPTON, T. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *FSE: Fast Software Encryption* (2004), Springer-Verlag, pp. 371–388.
- [16] TRIDGELL, A. Efficient algorithms for sorting and synchronization, Apr. 2000. PhD thesis, Australian National University.
- [17] VAN OORSCHOT, P., AND WIENER, M. Parallel collision search with cryptanalytic applications. *Journal of Cryptology* 12, 1 (1999), 1–28. Earlier version in ACM CCS ’94.
- [18] WANG, X., YAO, A., AND YAO, F. Recent improvements in finding collisions in sha-1. Presented at the rump session of CRYPTO ’05 by Adi Shamir.
- [19] WANG, X., AND YU, H. How to break MD5 and other hash functions. In *EUROCRYPT* (2005), vol. 3494 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 19–35.

A A Mathematical Note

This appendix is related to the paper, but not essential to the arguments made.

A further issue with Henson’s paper should be pointed out for purposes of clarity. Many of Henson’s claims are supported by calculation. However these calculations are sometimes not quite right: in section 3, she claims that the probability of one or more collisions among n outputs from a b -bit cryptographic hash function is $1 - (1 - 2^{-b})^n$. (Here we model the b -bit outputs as independent uniform random strings.) This is incorrect. In fact it greatly *underestimates* the collision probability. While it is true the probability that two b -bit random strings will not collide is $(1 - 2^{-b})$, we cannot raise this to the n and expect this to be the number of non-collisions among n total outputs since it neglects to count collisions across already-selected pairs.

The probability that n outputs of b -bits will contain a collision is $C(2^b, n) = 1 - (2^b - 1)/2^b \times (2^b - 2)/2^b \times \cdots \times (2^b - n + 1)/2^b$. It can be shown that, when $1 \leq n \leq 2^{(b+1)/2}$ we have $0.316n(n-1)/2^b \leq C(2^b, n) \leq 0.5n(n-1)/2^b$. This formula shows that the probability of a collision is *very* unlikely when n is well below the square root of 2^b , and then grows dramatically as n approaches this number. Indeed, it can be shown that the expected number of outputs needed before a collision occurs is $n \approx \sqrt{2^{b-1}\pi}$.

An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems

Paul Willmann, Scott Rixner, and Alan L. Cox
Rice University

{willmann, rixner, alc}@rice.edu

Abstract

As technology trends push future microprocessors toward chip multiprocessor designs, operating system network stacks must be parallelized in order to keep pace with improvements in network bandwidth. There are two competing strategies for stack parallelization. Message-parallel network stacks use concurrent threads to carry out network operations on independent messages (usually packets), whereas connection-parallel stacks map operations to groups of connections and permit concurrent processing on independent connection groups. Connection-parallel stacks can use either locks or threads to serialize access to connection groups. This paper evaluates these parallel stack organizations using a modern operating system and chip multiprocessor hardware.

Compared to uniprocessor kernels, all parallel stack organizations incur additional locking overhead, cache inefficiencies, and scheduling overhead. However, the organizations balance these limitations differently, leading to variations in peak performance and connection scalability. Lock-serialized connection-parallel organizations reduce the locking overhead of message-parallel organizations by using many connection groups and eliminate the expensive thread handoff mechanism of thread-serialized connection-parallel organizations. The resultant organization outperforms the others, delivering 5.4 Gb/s of TCP throughput for most connection loads and providing a 126% throughput improvement versus a uniprocessor for the heaviest connection loads.

1 Introduction

As network bandwidths continue to increase at an exponential pace, the performance of modern network stacks must keep pace in order to efficiently utilize that bandwidth. In the past, exponential gains in microprocessor

performance have always enabled processing power to catch up with network bandwidth. However, the complexity of modern uniprocessors will prevent such continued performance growth. Instead, microprocessors have begun to provide parallel processing cores to make up for the loss in performance growth of individual processor cores. For network servers to exploit these parallel processors, scalable parallelizations of the network stack are needed.

Modern network stacks can exploit either message-based parallelism or connection-based parallelism. Network stacks that exploit message-based parallelism, such as Linux and FreeBSD, allow multiple threads to simultaneously process different messages from the same or different connections. Network stacks that exploit connection-based parallelism, such as Dragonfly-BSD and Solaris 10 [16], assign each connection to a group. Threads may then simultaneously process messages as long as they belong to different connection groups. The connection-based approach can use either threads or locks for synchronization, yielding three major parallel network stack organizations: message-based (MsgP), connection-based using threads for synchronization (ConnP-T), and connection-based using locks for synchronization (ConnP-L).

The uniprocessor version of FreeBSD is efficient, but its performance falls short of saturating available network resources in a modern machine and degrades significantly as connections are added. Utilizing 4 cores, the parallel stack organizations can outperform the uniprocessor stack (especially at high connection loads), but each parallel stack organization incurs higher locking overhead, reduced cache efficiency, and higher scheduling overhead than the uniprocessor. MsgP outperforms the uniprocessor for almost all connection loads but experiences significant locking overhead. In contrast, ConnP-T has very low locking overhead but incurs significant scheduling overhead, leading to reduced performance compared to even the uniprocessor kernel for all

This work is supported in part by the Texas Advanced Technology Program under Grant No. 003604-0052-2003, by the NSF under Grant No. CCF-0546140, and by donations from AMD.

but the heaviest loads. ConnP-L mitigates the locking overhead of MsgP, by grouping connections so that there is little global locking, and the scheduling overhead of ConnP-T, by using the requesting thread for network processing rather than forwarding the request to another thread. This results in the best performance of all stacks considered, delivering stable performance of 5440 Mb/s for moderate connection loads and providing a 126% improvement over the uniprocessor kernel for large connection loads.

The following section further motivates the need for parallelized network stacks and discusses prior work. Section 3 then describes the parallel network stack architectures. Section 4 presents and discusses the results. Finally, Section 5 concludes the paper.

2 Background

Traditionally, uniprocessors have not been able to saturate the network with the introduction of each new Ethernet bandwidth generation, but exponential gains in uniprocessor performance have always allowed processing power to catch up with network bandwidth. However, the complexity of modern uniprocessors has made it prohibitively expensive to continue to improve processor performance at the same rate as in the past. Not only is it difficult to further increase clock frequencies, but it is also difficult to further improve the efficiency of complex modern uniprocessor architectures.

To further increase performance despite these challenges, industry has turned to single chip multiprocessors (CMPs) [12]. IBM, Sun, AMD, and Intel have all released dual-core processors [2, 15, 4, 8, 9]. Sun's Niagara is perhaps the most aggressive example, with 8 cores on a single chip, each capable of executing four threads of control [7, 10]. However, a CMP trades uniprocessor performance for additional processing cores, which should collectively deliver higher performance on parallel workloads. Therefore, the network stack will have to be parallelized extensively in order to saturate the network with modern microprocessors.

While modern operating systems exploit parallelism by allowing multiple threads to carry out network operations concurrently in the kernel, supporting this parallelism comes with significant cost [1, 3, 11, 13, 18]. For example, uniprocessor Linux kernels deliver 20% better end-to-end throughput over 10 Gigabit Ethernet than multiprocessor kernels [3].

In the mid-1990s, two forms of network processing parallelism were extensively examined: message-oriented and connection-oriented parallelism. Using message-oriented parallelism, messages (or packets) may be processed simultaneously by separate threads, even if those messages belong to the same connec-

tion. Using connection-oriented parallelism, messages are grouped according to connection, allowing concurrent processing of messages belonging to different connections.

Nahum *et al.* first examined message-oriented parallelism within the user-space *x*-kernel utilizing a simulated network device on an SGI Challenge multiprocessor [11]. This study found that finer grained locking around connection state variables generally degrades performance by introducing additional overhead and does not result in significant improvements in speedup. Rather, coarser-grained locking (with just one lock protecting all TCP state) performed best. They furthermore found that careful attention had to be paid to thread scheduling and lock acquisition ordering on the inbound path to ensure that received packets were not reordered during processing.

Yates *et al.* later examined a connection-oriented parallel implementation of the *x*-kernel, also utilizing a simulated network device and running on an SGI Challenge [18]. They found that increasing the number of threads to match the number of connections yielded the best results, even far beyond the number of physical processors. They proposed using as many threads as were supported by the system, which was limited to 384 at that time.

Schmidt and Suda compared message-oriented and connection-oriented network stacks in a modified version of SunOS utilizing a real network interface [14]. They found that with just a few connections, a connection-parallel stack outperforms a message-parallel one. However, they note that context switching increases significantly as connections (and processors) are added to the connection-parallel scheme, and that synchronization cost heavily affects the efficiency with which each scheme operates (especially the message-parallel scheme).

Synchronization and context-switch costs have changed dramatically in recent years. The gap between memory system and processing performance has become much greater, vastly increasing synchronization cost in terms of lost execution cycles and exacerbating the cost of context switches as thread state is swapped in memory. Both the need to close gap between Ethernet bandwidth and microprocessor performance and the vast changes in the architectural characteristics that shaped prior parallel network stack analyses motivate a fresh examination of parallel network stack architectures on modern parallel hardware.

3 Parallel Network Stack Architectures

Despite the conclusions of the 1990s, no solid consensus exists among modern operating system devel-

opers regarding efficient, scalable parallel network stack design. Current versions of FreeBSD and Linux incorporate variations of message parallelism within their network stacks. Conversely, the network stack within Solaris 10 incorporates a variation of connection-based parallelism [16], as does DragonflyBSD. Willmann *et al.* present a detailed description of parallel network stack organizations, and a brief overview follows [17].

3.1 Message-based Parallelism (MsgP)

Message-based parallel (MsgP) network stacks, such as FreeBSD, allow multiple threads to operate within the network stack simultaneously and permit these various threads to process messages independently. Two types of threads may perform network processing: one or more application threads and one or more inbound protocol threads. When an application thread makes a system call, that calling thread context is “borrowed” to carry out the requested service within the kernel. When the network interface card (NIC) interrupts the host, the NIC’s associated inbound protocol thread services the NIC and processes received packets “up” through the network stack.

Given these concurrent application and inbound protocol threads, FreeBSD utilizes fine-grained locking around shared kernel structures to ensure proper message ordering and connection state consistency. As a thread attempts to send or receive a message on a connection, it must acquire various locks when accessing shared connection state, such as the global connection hashtable lock (for looking up TCP connections) and per-connection locks (for both socket state and TCP state). This locking organization enables concurrent processing of different messages on the same connection.

Note that the inbound thread configuration described is not the FreeBSD 7 default. Normally parallel driver threads service each NIC and then hand off inbound packets to a single worker thread. That worker thread then processes the received packets “up” through the network stack. The default configuration limits the performance of MsgP, so is not considered in this paper. The thread-per-NIC model also differs from the message-parallel organization described by Nahum *et al.* [11], which used many more worker threads than interfaces. Such an organization requires a sophisticated scheme to ensure these worker threads do not reorder inbound packets, hence it is also not considered.

3.2 Connection-based Parallelism (ConnP)

To compare connection parallelism in the same framework as message parallelism, FreeBSD 7 was modified to support two variants of connection-based parallelism (ConnP) that differ in how they serialize TCP/IP pro-

cessing within a connection. The first variant assigns each connection to a protocol processing thread (ConnP-T), and the second assigns each connection to a lock (ConnP-L).

3.2.1 Thread Serialization (ConnP-T)

Connection-based parallelism using threads utilizes several kernel threads dedicated to protocol processing, each of which is assigned a subset of the system’s connections. At each entry point into the TCP/IP protocol stack, a request for service is enqueued for the appropriate protocol thread based on the TCP connection. Later, the protocol threads, which only carry out TCP/IP processing and are bound to a specific CPU, dequeue requests and process them appropriately. Because connections are uniquely and persistently assigned to a specific protocol thread, no per-connection state locking is required. These protocol threads implement both synchronous operations, for applications that require a return code, and asynchronous operations, for drivers that simply enqueue packets and then continue servicing the NIC.

The connection-based parallel stack uniquely maps a packet or socket request to a specific protocol thread by hashing the 4-tuple of remote IP address, remote port number, local IP address, and local port number. When the entire tuple is not yet defined (e.g., prior to port assignment during a `listen()` call), the corresponding operation executes on protocol thread 0 and may later migrate to another thread when the tuple becomes fully defined.

3.2.2 Lock Serialization (ConnP-L)

Connection-based parallelism using locks also separates connections into groups, but each group is protected by a single lock, rather than only being processed by a single thread. As in connection-based parallelism using threads, application threads entering the kernel for network service and driver threads passing up received packets both classify each request to a particular connection group. However, application threads then acquire the lock for the group associated with the given connection and then carry out the request with private access to any group-wide structures (including connection state). For inbound packet processing, the driver thread classifies each inbound packet to a specific group, acquires the group lock associated with the packet, and then processes the packet “up” through the network stack. As in the MsgP case, there is one inbound protocol thread for each NIC, but the number of groups may far exceed the number of threads.

This implementation of connection-oriented parallelism is similar to Solaris 10, which permits a network

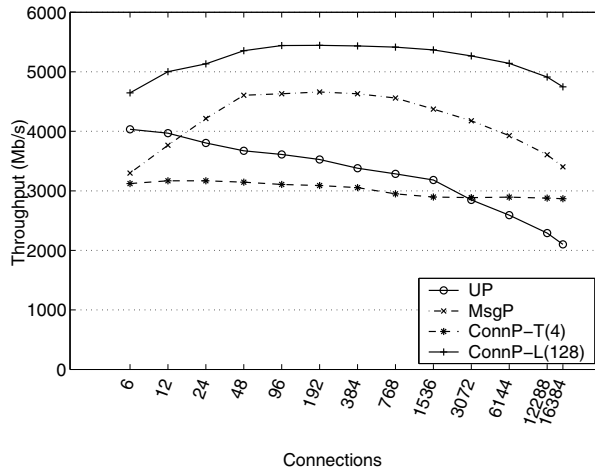


Figure 1: Aggregate network throughput.

operation to either be carried out directly after acquisition of a group lock or to be passed on to a worker thread for later processing. ConnP-L is more rigidly defined; application and inbound protocol threads always acquire exclusive control of the group lock.

4 Evaluation

The three competing parallelization strategies are implemented within the 2006-03-27 repository version of the FreeBSD 7 operating system for comparison on a 4-way SMP AMD Opteron system. The system consists of a Tyan S2885 motherboard, two dual-core Opteron 275 processors, two 1 GB PC2700 DIMMs per processor (one per memory channel), and three dual-port Intel PRO/1000-MT Gigabit Ethernet network interfaces spread across the motherboard's PCI-X bus segments. Data is transferred between the 4-way Opteron system and three client systems. The clients never limit the network performance of any experiment.

Each network stack organization is evaluated using a custom multithreaded, event-driven TCP/IP microbenchmark that distributes traffic across a configurable number of connections and uses zero-copy I/O. This benchmark manages connections using as many threads as there are processors. All experiments use the standard 1500-byte maximum transmission unit, and sending and receiving socket buffers are 256 KB each.

Figure 1 depicts the aggregate throughput across all connections when executing the parallel TCP benchmark utilizing various configurations of FreeBSD 7. "UP" is the uniprocessor version of the FreeBSD kernel running on a single core of the Opteron server; all other kernel configurations use all 4 cores. "MsgP" is the multiprocessor MsgP kernel described in Section 3.1. MsgP uses a lock per connection. "ConnP-T(4)" is the multipro-

OS Type	6 conns	192 conns	16384 conns
MsgP	89	100	100
ConnP-L(4)	60	56	52
ConnP-L(8)	51	30	26
ConnP-L(16)	49	18	14
ConnP-L(32)	41	10	7
ConnP-L(64)	37	6	4
ConnP-L(128)	33	5	2

Table 1: Percentage of lock acquisitions for global TCP/IP locks that do not succeed immediately.

cessor ConnP-T kernel described in Section 3.2.1, using 4 kernel protocol threads for TCP/IP stack processing that are each pinned to a different core. "ConnP-L(128)" is the multiprocessor ConnP-L kernel described in Section 3.2.2. ConnP-L(128) divides the connections among 128 locks within the TCP/IP stack.

The figure shows that the uniprocessor kernel performs well with a small number of connections, achieving a bandwidth of 4034 Mb/s with only 6 connections. However, total bandwidth decreases as the number of connections increases. MsgP achieves 82% of the uniprocessor bandwidth at 6 connections but quickly ramps up to 4630 Mb/s, holding steady through 768 connections and then decreasing to 3403 Mb/s with 16384 connections. ConnP-T(4) achieves close to its peak bandwidth of 3123 Mb/s with 6 connections and provides approximately steady bandwidth as the number of connections increase. Finally, the ConnP-L(128) curve is shaped similar to that of MsgP, but its performance is larger in magnitude and always outperforms the uniprocessor kernel. ConnP-L(128) delivers steady performance around 5440 Mb/s for 96–768 connections and then gradually decreases to 4747 Mb/s with 16384 connections. This peak performance is roughly the peak TCP throughput deliverable by the three dual-port Gigabit NICs.

Figure 1 shows that using 4 cores, ConnP-L(128) and MsgP outperform the uniprocessor FreeBSD 7 kernel for almost all connection loads. However, the speedup is significantly less than ideal and is limited by (1) locking overhead, (2) cache efficiency, and (3) scheduling overhead. The following subsections will explain how these issues affect the parallel implementations of the network stack.

4.1 Locking Overhead

Both lock latency and contention are significant sources of overhead within parallelized network stacks. Within the network stack, there are both global and individual locks. Global locks protect hash tables that are used to access individual connections, and individual locks protect only one connection. A thread must acquire a global

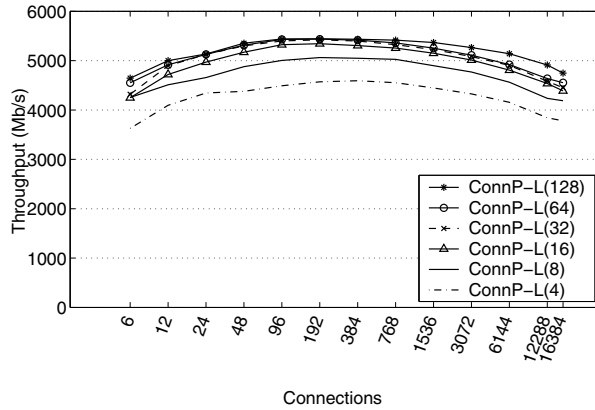


Figure 2: Aggregate network throughput for ConnP-L as the number of locks is varied.

lock to look up and access an individual lock. During contention for these global locks, other threads are blocked from entering the associated portion of the network stack, limiting parallelism.

Table 1 depicts global TCP/IP lock contention, measured as the percentage of lock acquisitions that do not immediately succeed because another thread holds the lock. ConnP-T is omitted from the table because it eliminates global TCP/IP locking completely. The MsgP network stack experiences significant contention for global TCP/IP locks. The Connection Hashtable lock protecting individual Connection locks is particularly problematic. Lock profiling shows that contention for Connection locks decreases with additional connections, but that the cost for contention for these locks increases because as the system load increases, they are held longer. Hence, when a Connection lock is contended (usually between the kernel's inbound protocol thread and an application's sending thread), a thread blocks longer holding the global Connection Hashtable lock, preventing other threads from making progress.

Whereas the MsgP stack relies on repeated acquisition of the Connection Hashtable and Connection locks, ConnP-L stacks can also become bottlenecked if a single connection group becomes highly contended. Table 1 shows the contention for the Network Group locks for ConnP-L stacks as the number of network groups is varied. Though ConnP-L(4)'s Network Group lock contention is high at over 50% for all connection loads, increasing the number of groups to 128 reduces contention from 52% to just 2% for the heaviest load. Figure 2 shows the effect that increasing the number of network groups has on aggregate throughput. As is suggested by reduced Network Group lock contention, throughput generally increases as groups are added, although with diminishing returns.

OS Type	6 conns	192 conns	16384 conns
UP	1.83	4.08	18.49
MsgP	37.29	28.39	40.45
ConnP-T(4)	52.25	50.38	51.39
ConnP-L(128)	28.91	26.18	40.36

Table 2: L2 Data cache misses per KB of transmitted data.

OS Type	6 conns	192 conns	16384 conns
UP	481.77	440.20	422.84
MsgP	2904.09	1818.22	2448.10
ConnP-T(4)	3487.66	3602.37	4535.38
ConnP-L(128)	2135.26	923.93	1063.65

Table 3: Cycles of scheduler overhead per KB of transmitted data.

4.2 Cache Behavior

Table 2 shows the number of L2 data cache misses per KB of payload data transmitted, effectively normalizing cache hierarchy efficiency to network bandwidth. The uniprocessor kernel incurs very few cache misses relative to the multiprocessor configurations because of the lack of migration. As connections are added, the associated increase in connection state stresses the cache and directly results in increased cache misses [5, 6].

The parallel network stacks incur significantly more cache misses per KB of transmitted data because of data migration and lock accesses. Surprisingly, ConnP-T(4) incurs the most cache misses despite each thread being pinned to a specific processor. While thread pinning can improve locality by eliminating migration of connection metadata, frequently updated socket metadata is still shared between the application and protocol threads, which leads to data migration and a higher cache miss rate.

4.3 Scheduler Overhead

The ConnP-T kernel trades the locking overhead of the ConnP-L and MsgP kernels for scheduling overhead. Network operations for a particular connection must be scheduled onto the appropriate protocol thread. Figure 1 showed that this results in stable, but low total bandwidth as connections scale for ConnP-T. Conversely, ConnP-L minimizes lock contention with additional groups and reduces scheduling overhead since messages are not transferred to protocol threads. This results in consistently better performance than the other parallel organizations.

Table 3 shows scheduler overhead normalized to network bandwidth, measured in cycles spent managing the scheduler and scheduler synchronization per KB of payload data transmitted. Though MsgP experiences less scheduling overhead as the number of connections in-

crease and threads aggregate more work, locking overhead within the threads quickly negate the scheduler advantage. In contrast, the scheduler overhead of ConnP-T remains high, corresponding to relatively low bandwidth. This highlights that ConnP-T's thread-based serialization requires efficient inter-thread communication to be effective. In contrast, ConnP-L exhibits stable scheduler overhead that is much lower than ConnP-T and MsgP, contributing to its higher throughput. ConnP-L does not require a thread handoff mechanism and its low lock contention compared to MsgP results in fewer context switches from threads waiting for locks.

5 Conclusions

Network performance is increasingly important in all types of modern computer systems. Furthermore, architectural trends are pushing future microprocessors away from uniprocessor designs and toward architectures that incorporate multiple processing cores and/or thread contexts per chip. This trend necessitates the parallelization of the operating system's network stack. This paper evaluates message-based and connection-based parallelism within the network stack of a modern operating system. Further results and analysis are available in a technical report [17].

The uniprocessor version of the FreeBSD operating system performs quite well, but its performance degrades as additional connections are added. Though the MsgP, ConnP-T, and ConnP-L parallel network stacks can outperform the uniprocessor when using 4 cores, none of these organizations approach perfect speedup. This is caused by the higher locking overhead, poor cache efficiency, and high scheduling overhead of the parallel organizations. While MsgP can outperform a uniprocessor by 31% on average and by 62% for the heaviest connection loads, the enormous locking overhead incurred by such an approach limits its performance and prevents it from saturating available network resources. In contrast, ConnP-T eliminates intrastack locking completely by using thread serialization but incurs significant scheduling overhead that limits its performance to less than that of the uniprocessor kernel for all but the heaviest connection loads. ConnP-L mitigates the locking overhead of MsgP, by grouping connections to reduce global locking, and the scheduling overhead of ConnP-T, by using the requesting thread for network processing rather than invoking a network protocol thread. This results in good performance across a wide range of connections, delivering 5440 Mb/s for moderate connection loads and achieving a 126% improvement over the uniprocessor kernel when handling large connection loads.

References

- [1] BJÖRKMAN, M., AND GUNNINGBERG, P. Performance modeling of multiprocessor implementations of protocols. *IEEE/ACM Transactions on Networking* (June 1998).
- [2] DIFENDORFF, K. Power4 focuses on memory bandwidth. *Microprocessor Report* (Oct. 1999).
- [3] HURWITZ, J., AND FENG, W. End-to-end performance of 10-gigabit Ethernet on commodity systems. *IEEE Micro* (Jan./Feb. 2004).
- [4] KAPIL, S., MCGHAN, H., AND LAWRENDRA, J. A chip multithreaded processor for network-facing workloads. *IEEE Micro* (Mar./Apr. 2004).
- [5] KIM, H., AND RIXNER, S. Performance characterization of the FreeBSD network stack. Tech. Rep. TR05-450, Rice University Computer Science Department, June 2005.
- [6] KIM, H., AND RIXNER, S. TCP offload through connection handoff. In *Proceedings of EuroSys* (Apr. 2006).
- [7] KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro* (Mar./Apr. 2005).
- [8] KREWELL, K. UltraSPARC IV mirrors predecessor. *Microprocessor Report* (Nov. 2003).
- [9] KREWELL, K. Double your Operons; double your fun. *Microprocessor Report* (Oct. 2004).
- [10] KREWELL, K. Sun's Niagara pours on the cores. *Microprocessor Report* (Sept. 2004).
- [11] NAHUM, E. M., YATES, D. J., KUROSE, J. F., AND TOWSLEY, D. Performance issues in parallelized network protocols. In *Proceedings of the Symposium on Operating Systems Design and Implementation* (Nov. 1994).
- [12] OLUKOTUN, K., NAYFEH, B. A., HAMMOND, L., WILSON, K., AND CHANG, K. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1996).
- [13] ROCA, V., BRAUN, T., AND DIOT, C. Demultiplexed architectures: A solution for efficient STREAMS-based communication stacks. *IEEE Network* (July 1997).
- [14] SCHMIDT, D. C., AND SUDA, T. Measuring the performance of parallel message-based process architectures. In *Proceedings of the INFOCOM Conference on Computer Communications* (Apr. 1995).
- [15] TENDLER, J. M., DODSON, J. S., J. S. FIELDS, J., LE, H., AND SINHARAY, B. Power4 system architecture. *IBM Journal of Research and Development* (Jan. 2002).
- [16] TRIPATHI, S. FireEngine—a new networking architecture for the Solaris operating system. White paper, Sun Microsystems, June 2004.
- [17] WILLMANN, P., RIXNER, S., AND COX, A. L. An evaluation of network stack parallelization strategies in modern operating systems. Tech. Rep. TR06-872, Rice University Computer Science Department, Apr. 2006.
- [18] YATES, D. J., NAHUM, E. M., KUROSE, J. F., AND TOWSLEY, D. Networking support for large scale multiprocessor servers. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (May 1996).

Disk Drive Level Workload Characterization

Alma Riska
Seagate Research
1251 Waterfront Place
Pittsburgh, PA 15222
Alma.Riska@seagate.com

Erik Riedel
Seagate Research
1251 Waterfront Place
Pittsburgh, PA 15222
Erik.Riedel@seagate.com

Abstract

In this paper, we present a characterization of disk drive workloads measured in systems representing the enterprise, desktop, and consumer electronics environments. We observe that the common characteristics across all traces are disk drive idleness and workload burstiness. Our analysis shows that the majority of characteristics, including request disk arrival rate, response time, service time, WRITE performance, and request size, are environment dependent. However, characteristics such as READ/WRITE ratio and access pattern are application dependent.

1 Introduction

Hard disk drives have become the *only* versatile form of persistent storage that offers reliability, high performance, and a wide range of form factors and capacities, which best suits the needs of highly dynamic and ever changing computing environments. Currently, storage solutions are in demand not only from traditional computer systems, such as the high-end servers, desktop and notebook PCs, but also from the very fast growing sector of consumer electronics.

We stress that it is critical to distinguish the environment where disk drives operate, because traditionally different *hardware* is used in different environments. For example the high performing SCSI disk drives are used in the enterprise environment while ATA/IDE disk drives are used in the desktop/notebook and consumer electronics environments [2]. The understanding of per-environment workload characteristics guides the decision making to support a specific application with the right hardware.

Because storage is the slowest processing unit in a computer system, accurate IO workload characterization allows for efficient storage optimization [1, 6, 14], which targets the specific requirements of the computing envi-

ronment. The focus of IO workload characterization has always been on large high-end multi-user computer systems [11, 13], because of the criticality of the applications running there, such as databases, Internet services, and scientific applications. Nevertheless, behavior of file systems is evaluated across various environments in [12]. The file system workload in personal computers is characterized in [16] and particularly for Windows NT workstations in [15]. With the workload characterization that we present here, we intend to cover all computing environments where modern disk drives are used, i.e., enterprise, desktop PCs, and consumer electronics. Different from all previous work [11, 12, 13] on tracing the storage system, we do not need to modify the software stack, because the measurements are conducted outside the system by attaching a SCSI or IDE analyzer to the IO bus and intercepting the electrical signals.

Our traces are measured at the disk level or the RAID-array controller level in various systems, including web servers, e-mail servers, game consoles, and personal video recorders (PVR). This set of disk level traces represents the majority of IO workloads. Traces are measured in both production and controlled systems. In controlled systems, only the scenario under which the application runs is controlled. All measurements were taken during the 2004 calendar year in state-of-the-art systems at the time.

Our evaluation tends to agree with previous work [5, 12, 13] on the application-dependent nature of IO workload characteristics. We observe that READ/WRITE ratio, access pattern, and WRITE traffic handling vary by application. However, the majority of characteristics vary only by environment. Environment dependent characteristics include request disk arrival rate, service time, response time, sequentiality, and average length of idle intervals. We observe that CE tends to be a more variable environment when it comes to workload characteristics than the desktop or enterprise ones. Despite workload dependencies on specific applications and environ-

ments, there are characteristics that describe the overall IO workload that remain consistent through applications and environments. One of them is workload burstiness. We observe that disk level workload is bursty similarly to network [10], Web server [4], and file system traffic [8]. Because the evaluation of disk drive workload burstiness is critical for devising accurate disk drive synthetic workload generators [5] and predictions of storage system behavior [7], we quantify it and show that burstiness is exhibited at the disk drive level in almost all computing environments.

The rest of this paper is organized as follows. In Section 2, we explain the measurement environments and give the trace format. In Section 3, we present general characterization of the trace data. Section 4 presents a detailed analysis of the interarrival process captured by our traces. In Section 5, we discuss the characteristics of the service process in our traces. We conclude with Section 6 where we summarize our results.

2 Measurement Environment

In this paper, we characterize disk drive workloads with respect to different computing environments, categorized based on their complexity, applications, and performance requirements as follows:

- *Enterprise*, which includes high-end computer systems, where the storage system often consists of RAID arrays and supports applications such as web, databases, and e-mail servers.
- *Desktop*, which includes personal computers with single disk drives, that support single-user applications.
- *Consumer Electronics - CE*, which includes consumer electronic devices that have a disk drive for their storage needs. CE devices include PVRs, game consoles, MP3 players, and digital cameras.

Because of the Non-Disclosure Agreements that are in place for the traces evaluated in this paper, we will not explain in detail the measurement systems. However we note that our enterprise traces are measured in production systems. This means that the system owner allowed us to conduct the measurement while the system was running a daily routine. Except the Web server, all other enterprise systems are multi-disk systems configured in RAID arrays. All measured enterprise systems consist of SCSI disk drives. In cases when the trace length is less than 24 hours, the measurement is conducted during business hours. Actually during the 25 hours of the E-mail trace, the highest burst of load is during the nightly backup activity. Currently, we have not conducted measurements in an enterprise database system, and this is an application category that is lacking in our trace collection.

The desktop traces are measured on employee' (engineer') PCs operating under Windows or Linux, while they run their daily applications. The CE traces are measured in controlled systems (i.e., the scenario of the application is set by the engineer performing the measurement). We have two PVR traces measured on the PVRs of two different vendors. Trace "PVR A" runs overnight; it records 2-hour shows continuously, plays back the shows periodically, and in the same time conducts media scrubbing. Similarly "PVR B" plays back and records simultaneously in a span of 3 hours. The "Game Console" trace is measured on a game console while a game is played. The "MP3" trace is measured on a networked system with 3 players and songs that span in a 10-20GB LBA range. All measured desktop and CE systems consist of ATA/IDE disk drives

All traces are measured using a SCSI or IDE analyzer that intercepts the IO bus electrical signals and stores them. The signals are decoded at a later time to generate the final traces. The choice of the analyzer enables trace collection without modifying the software stack of the targeted system and does not affect system performance.

Traces record for each request the arrival time and the departure time in a scale of 1/100 of a millisecond. Since the average service time of a single disk drive request is several milliseconds, the granularity of the arrival and departure times is accurate for our evaluation purposes. In addition, each trace record contains request length in bytes, the first logical block number of the requested data, the type of each request, (i.e., READ or WRITE), the disk ID (when measurements are performed in an array of disks), and the queue length at the disk. The length of the traces varies from one hour to several hours and the number of requests in the traces ranges from several thousands to a few millions.

3 General Analysis

From the extended set of traces that we have, we selected a few representative ones to make the presentation easier. Table 1 lists all traces that we evaluate in this paper and their main characteristics such as the number of disks in the system, trace length in hours, number of requests in the trace, the READ/WRITE ratio, IO bus idleness, the average length of idle intervals, the average response time at the disk drive, and the average and maximum queue length at the drive (as seen by each arriving request).

An important observation from Table 1 is that disks are idle. Yet bus idleness (which is measured in our traces) does not mean that there are no outstanding requests in the IO system. For example, when disk queue depth is one then queuing happens at the file system and often the bus remains idle for less than 1 ms between

Trace	Enviro- nment	No of Disks	Length	# of Reqs.	R/W %	Bus Idle	Avg. Bus Idle Int	Avg. Resp. Time	Avg./Max QL
Web	ENT	1	7.3 hrs	114,814	44/56	96%	274 ms	13.06 ms	1 / 16
E-mail	ENT	42	25 hrs	1,606,434	99/1	92%	625 ms	13.28 ms	3 / 9
Software Dev.	ENT	42	12 hrs	483,563	88/12	94%	119 ms	8.62 ms	2 / 7
User Accounts	ENT	42	12 hrs	168,148	87/13	98%	183 ms	12.82 ms	3 / 8
Desktop 1	DESK	1	21 hrs	146,248	52/48	99%	1000 ms	3.08 ms	1 / 1
Desktop 2	DESK	1	18 hrs	159,405	15/85	99%	506 ms	2.63 ms	1 / 1
Desktop 3	DESK	1	24 hrs	29,779	44/56	99%	402 ms	2.64 ms	1 / 1
PVR A	CE	1	20 hrs	880,672	95/5	89%	72 ms	9.77 ms	1 / 1
PVR B	CE	1	2.8 hrs	138,155	54/46	82%	60 ms	8.20 ms	1 / 1
MP3	CE	1	2.2 hrs	40,451	69/31	18%	37 ms	5.71 ms	1 / 1
Game Console	CE	1	1.4 hrs	33,076	83/17	95%	142 ms	1.08 ms	1 / 1

Table 1: General characteristics. Traces are identified by the application, dedicatedly supported by the storage system.

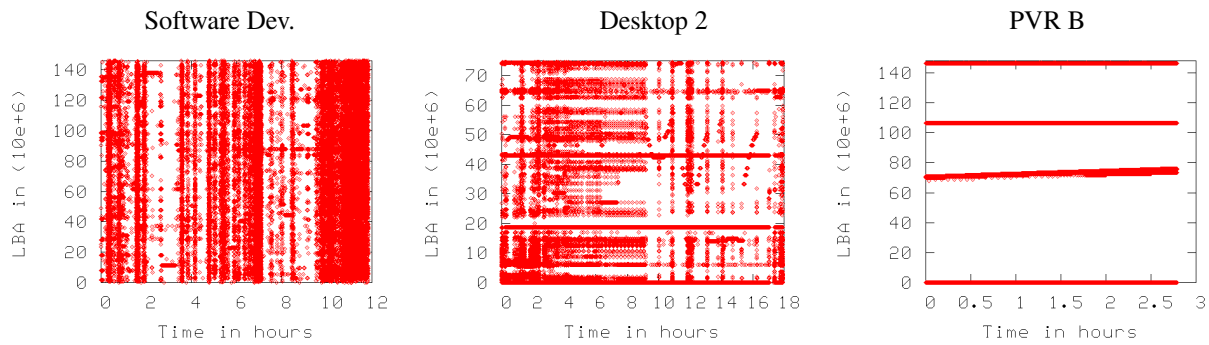


Figure 1: Access pattern (LBA accessed across time). Horizontal lines indicate sequential disk accesses.

one disk request completion and a new disk request arrival. Identifying idleness at the disk drive is important when it comes to scheduling background activities such as scrubbing, flushing the cache, and powering down the disk (to save energy). Note that different background activities have different requirements for idleness. While scrubbing and flushing the cache can be completed in intervals of tens to hundreds of milliseconds, powering down should be triggered for idle intervals of more than several seconds or minutes. Our traces indicate that the percentage of idle intervals larger than 200 ms account for, in average, 20%, 50%, and 35% of all idle intervals of at least 1 ms in the enterprise, desktop, and CE environments, respectively.

The average response time (see Table 1) in all systems is only several milliseconds. For the enterprise traces, response time is the sum of the queuing time and service time (as many as 16 requests are queued at the measured SCSI drives). For the desktop and CE traces, because there is no queuing, the response time approximates service time at the disk drive, reasonably well.

The access pattern, (i.e., the request position), is among the most important characteristics of disk drive

workloads, because it is related to the disk service process. In general, it is accepted that enterprise and desktop environments operate under mostly random workloads, (i.e., requests are distributed uniformly on (and across) disk surfaces). Random workloads have high service demands, because the disk arm has to seek from one track to the next. On the other hand, sequential IO workloads, often associated with video/audio streaming in various CE devices, require only moderate head movements and, consecutively, have low service demands. Figure 1 depicts the access patterns for three of the traces of Table 1. As expected, the access pattern for enterprise is more random than for desktop and the CE is highly sequential. Note that the range of accessed LBAs spans throughout the available space at the disk, indicating that (at least for enterprise and desktop) the disks operate close to their capacity and most data is accessed during the measurement period

We observe that in enterprise systems the degree of sequentiality between READs and WRITEs is different. WRITEs are more sequential than READs because various caches in the IO path coalesce small WRITEs for better performance. We measure the degree of sequen-

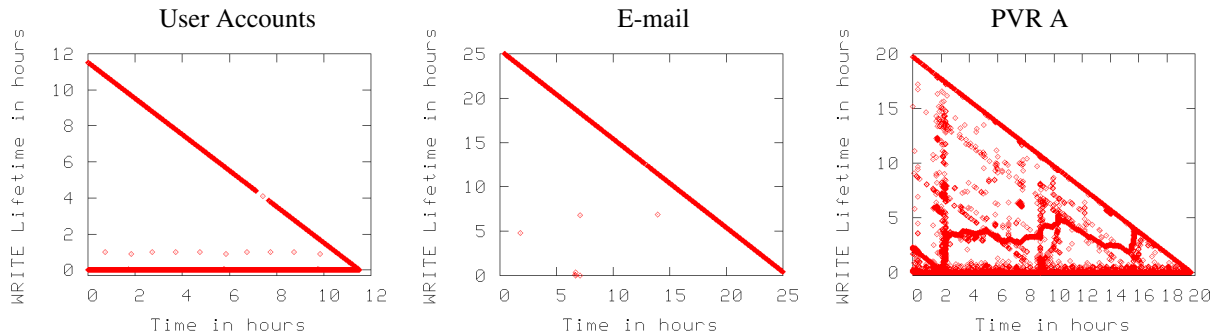


Figure 3: Lifetime of each WRITE across time. The non-overwritten WRITES fall on the diagonal of the plots.

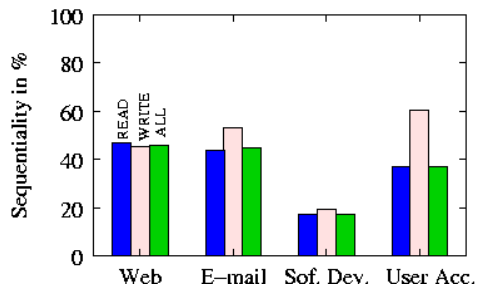


Figure 2: Sequentiality of traffic in enterprise environment. The measure is the portion of fully sequential requests to the total number of requests for READs, WRITES, and both.

tiality as the portion of requests from the IO stream that is fully sequential (i.e. end LBA of a request is the start LBA for the consecutive one) and show it in Figure 2 only for enterprise traces. CE traces have a sequentiality twice as much as enterprise traces and omitted here due to lack of space.

Table 2 shows statistics on request size. Note that the mean of request sizes varies while the variability of them (measured via the coefficient of variation) is consistently low. Across all traces, the common request size is 4KB (third column in Table 2), except the video streaming and game console which issue 128KB requests. The largest measured request size is 128KB although in both Software Development and E-mail traces there are occurrences (5 in total per trace) of requests larger than 128KB (but no more than 244KB).

In designing storage systems, particularly the high-end ones, considerable effort goes to handling WRITE traffic efficiently because it is related to both performance and reliability. NVRAM, which safely stores the data even during power outages and system crashes, is seen as an attractive feature in various levels of the IO path, to optimize handling of WRITE traffic [13]. In Figure 3, we present WRITES lifetime as a function of time for three

Trace	Mean	CV	Common Req. Sizes	
			1st	2nd
Web	16K	1.4	4K(55%)	64K(17%)
E-mail	16K	1.2	4K(60%)	8K(8 %)
Software Dev.	20K	1.2	4K(40%)	16K(17%)
User Accounts	23K	1.3	4K(43%)	8K(11%)
Desktop 1	10K	1.3	4K(56%)	.5K(20%)
Desktop 2	23K	1.1	4K(41%)	64K(24%)
Desktop 3	13K	1.3	4K(34%)	.5K(20%)
PVR A	5K	2.4	4K(30%)	1K(22%)
PVR B	42K	1.3	128K(29%)	8K(21%)
MP3 Player	12K	1.0	4K(62%)	32K(26%)
Game Console	54K	0.9	128K(27%)	64K(12%)

Table 2: The mean and the coefficient of variation of request sizes. The third and the fourth column show the two most common sizes in KB and their respective portion for each trace.

traces from Table 1. For each WRITE request, we note if any of its blocks are rewritten again for the duration of the trace. If a set of blocks are not re-written then their WRITE lifetime is the remaining of trace duration. Non re-written blocks fall on the diagonals of the plots in Figure 3. We observe that the enterprise environments such as the “E-mail” trace does not overwrite any data written during the measurement period. This indicates that the upper layers of the IO path effectively optimize WRITE traffic to the disk drive. The “User Accounts” trace is expected to consists of requests that update file system metadata and have stringent consistency requirements, (i.e., the file system forces the data to the disk rather than keeping it in any of the caches of the IO path). This system behavior causes a portion of WRITE traffic to get re-written in short time intervals. In the CE traces (see the “PVR A” plot in Figure 3), there is no NVRAM to optimize WRITE traffic, hence there are many re-written blocks during the traces.

Trace	Interarrival times		Service times	
	Mean	CV	Mean	CV
Web	229.6	6.3	6.64	0.69
E-mail	56.9	9.0	5.59	0.75
Software Dev.	88.0	12.3	6.34	0.84
User Accounts	246.6	3.8	6.10	0.74
Desktop 1	509.8	17.7	3.08	2.22
Desktop 2	405.0	7.0	2.63	1.98
Desktop 3	2882.2	19.2	2.64	1.97
PVR A	80.6	8.2	9.77	0.55
PVR B	72.7	1.0	8.20	1.24
MP3 Player	195.2	2.1	5.71	2.38
Game Console	148.4	3.2	1.08	0.83

Table 3: The mean and the coefficient of variation of interarrival and service times for our traces.

4 Interarrival Times

We present the general statistical properties of the request interarrival times, (i.e., mean and coefficient of variation) in Table 3. Note that enterprise and CE traces have higher average arrival rates than the desktop ones. The interarrival times are consistently variable with CVs being as high as 19, except for the CE traces. We expect CE applications, which mostly stream video/audio, to have low variability in the interarrival times.

We also focus on burstiness of arrivals and estimate the Hurst parameter, which is widely used to quantify long-range dependence and burstiness [3, 10]. We use the Selfis [9] tool to compute the Hurst parameter using two different techniques, (i.e., the Aggregate variance, and the Peridogram). Details on these two techniques can be found in [3]. We present our results in Table 4. Note that a Hurst parameter value of 0.5 or larger indicates long-range dependence [3] in the time series (in our case the series of interarrival times). We conclude that interarrival times at the disk drive are long-range dependent, because all values in Table 4 are larger than 0.5. One of the direct consequences of long-range dependent interarrival times is on the queuing behavior and saturation, which happens faster under long-range dependent than independent interarrival times.

5 Service Process

In this section, we focus on the service process at the disk drive for traces of Table 1. Recall that in our traces, we record only the arrival time and the departure time per request. Because there is no queuing for desktop and CE traces, the difference between departure and arrival times accurately approximates the service time at the disk drive. For the SCSI traces where queuing is

Trace	Interarrival time		Seek Distance	
	Agg. Var.	Peridogram	Agg. Var.	Peridogram
Web Server	0.81	0.672	0.89	0.972
E-mail	0.83	0.727	0.76	0.734
User Acc.	0.75	0.782	0.78	0.758
Soft.Dev.	0.79	0.498	0.75	0.885
Desktop 1	0.71	0.593	0.78	1.002
Desktop 2	0.84	0.675	0.82	0.938
Desktop 3	0.73	0.640	0.88	0.963
PVR A	0.86	0.614	0.87	0.873
PVR B	0.54	0.577	0.59	0.243
MP3 Player	0.83	0.928	0.74	0.976
Game Console	0.82	0.842	0.87	1.248

Table 4: Hurst parameters of interarrival times and seek distances computed via the Aggregate Variance and Peridogram methods.

present at the disk, we estimate the characteristics of the service process by using the data of only those requests that find an empty queue at the disk upon arrival. Note that such an approximation of the service time process is not far from the real one, because the load in our enterprise traces is light and most busy periods have only one request.

Table 3 shows the mean and the coefficient of variation for the estimated service times of our traces. Observe that the average service time is several milliseconds. Note that a CV less than 1 indicates that a process has lower variability than the well-behaved exponential distribution whose CV is 1. Enterprise and desktop environments perform consistently within the environment. Desktop traces have lower service times and higher CVs than enterprise traces because, first, our desktop traces are more sequential than enterprise ones, and, second, because desktop drives operate with WRITE-back cache enabled and enterprise drives operate with WRITE-through cache enabled. The latter is easily extracted from the traces, because the majority of WRITES in the desktop traces take less than a millisecond to complete. The CE environment shows inconsistency in the mean and CV of service times across traces, making the CE service process application dependent.

We also analyze the dependence structure of request positions, as a way to understand dependencies in the service process and access patterns at the disk drive level. We compute the Hurst parameter for seek (LBA) distances between consecutive requests and present the results in Table 4. Recall that for each request in the trace, we record the start LBA and the request size. Hence, we compute the seek distance as the difference between the start-LBA of each requests and the end-LBA of the

previous request.

Note that the seek distances in all traces exhibit extreme long-range dependence. Quantitatively, seek distances exhibit stronger long-range dependence than the interarrival times (shown in Table 4 as well). Such behavior again confirms that locality is an inherent characteristic of disk drive workloads [13]. Although, enterprise workloads are more random than the desktop or CE ones, in enterprise systems, there are several IO schedulers and caches in the layered IO path that order and coalesce requests such that the logical seek distances between consecutive requests is minimized.

6 Conclusions

In this paper, we characterized disk drive workloads in three different computing environments, (i.e., enterprise, desktop, and consumer electronics). Our evaluation agrees with previous work, on the application-dependent nature of IO workload characteristics. We observe that READ/WRITE ratio, access pattern, handling of WRITES vary by specific applications. However, the majority of characteristics vary only by environment. Environment dependent characteristics include the length of idle intervals, request arrival rate, request disk service time and response time, workload sequentiality, WRITE performance, and request size. More importantly, there are characteristics of the overall IO workload that do remain consistent through applications and environments. One of particular note is workload burstiness (i.e., long-range dependence). We observe that disk level workloads, in particular, request interarrival times and request seek distances are long-range dependent. Long-range dependence, as a measure of temporal locality in a time series, has a variety of consequences in particular when it comes to predict overall system and specific resource saturation. As a result, burstiness should be taken under consideration when designing new storage systems, and resource management policies at various layers of the IO path.

Acknowledgments

We would like to thank James Dykes, Jinglei Li, and Scott Borton for all their efforts collecting and parsing these traces and making them available to us. We would like to thank Kimberly Keeton for excellent shepherding of our paper. Her comments and the comments of the anonymous reviewers greatly improved the presentation of this paper.

References

- [1] ALVAREZ, G., KEETON, K., RIEDEL, E., AND UYSAL, M. Characterizing data-intensive workloads on modern disk arrays. In *Proceedings of the Workshop on Computer Architecture Evaluation using Commercial Workloads* (2001).
- [2] ANDERSON, D., DYKES, J., AND RIEDEL, E. SCSI vs. ATA - More than an interface. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies, (FAST'03)* (San Francisco, CA, 2003).
- [3] BERAN, J. *Statistics for Long-Memory Processes*. Chapman & Hall, New York, 1994.
- [4] CROVELLA, M. E., AND BESTAVROS, A. Self-similarity in world wide web traffic: evidence and possible causes. In *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (1996), ACM Press, pp. 160–169.
- [5] GANGER, G. R. Generating representative synthetic workloads: an unsolved problem. In *Proceedings of Computer Measurement Group (CMG) Conference* (Dec. 1995), pp. 1263–1269.
- [6] GOMEZ, M. E., AND SANTONJA, V. A new approach in the modeling and generation of synthetic disk workload. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems; MASCOTS'00* (2000), IEEE Computer Society, pp. 199–207.
- [7] GOMEZ, M. E., AND SANTONJA, V. On the impact of workload burstiness on disk performance. In *Workload characterization of emerging computer applications* (2001), Kluwer Academic Publishers, pp. 181–201.
- [8] GRIBBLE, S. D., MANKU, G. S., ROSELLI, D., BREWER, E. A., GIBSON, T. J., AND MILLER, E. L. Self-similarity in file systems. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems* (1998), pp. 141–150.
- [9] KARAGIANNIS, T., FALOUTSOS, M., AND MOLLE, M. A user-friendly self-similarity analysis tool. *Special Section on Tools and Technologies for Networking Research and Education, ACM SIGCOMM Computer Communication Review* 33, 3 (2003), 81–93.
- [10] LELAND, W. E., TAQQU, M. S., WILLINGER, W., AND WILSON, D. V. On the self-similar nature of Ethernet traffic. *IEEE/ACM Transactions on Networking* 2 (1994), 1–15.
- [11] OUSTERHOUT, J. K., COSTA, H. D., HARRISON, D., KUNZE, J. A., KUPFER, M. D., AND THOMPSON, J. G. A trace-driven analysis of the unix 4.2 bsd file system. In *SOSP* (1985), pp. 15–24.
- [12] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. A comparison of file systems workloads. In *Proceedings of USENIX Technical Annual Conference* (2000), pp. 41–54.
- [13] RUEMLER, C., AND WILKES, J. Unix disk access patterns. In *Proceedings of the Winter 1993 USENIX Technical Conference* (1993), pp. 313–323.
- [14] RUEMLER, C., AND WILKES, J. An introduction to disk drive modeling. *Computer* 27, 3 (1994), 17–28.
- [15] VOGELS, W. File system usage in windows nt 4.0. In *Proceedings of the ACM Symposium on Operation Systems Principals (SOSP)* (1999), pp. 93–109.
- [16] ZHOU, M., AND SMITH, A. J. Analysis of personal computer workloads. In *Proceedings of the International symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (1999), pp. 208–217.

Towards a Resilient Operating System for Wireless Sensor Networks

Hyoseung Kim and Hojung Cha

*Department of Computer Science, Yonsei University
Seodaemun-gu, Shinchon-dong 134, Seoul 120-749, Korea
{hskim, hjcha}@cs.yonsei.ac.kr*

Abstract

Active research has recently been conducted on large scale wireless sensor networks, especially network management and maintenance, but the technique for managing application errors on MMU-less sensor node devices has not been seriously considered. This paper presents a resilient operating system mechanism for wireless sensor networks. The proposed mechanism separates the kernel from the user execution area via dual mode operation, and the access violation of applications is controlled by static/dynamic code checking. The experiment results on a common sensor node show that the proposed mechanisms effectively protect the system from errant applications.

1. Introduction

Wireless sensor networks normally consist of battery-operated, memory-limited and low performance node devices. Although the research on radio communication techniques and system software developments for resource constraint devices has recently been active [1][2], little effort has been expended on the reliability issue on sensor network systems running on MMU-less devices. Application errors on sensor nodes can affect the entire system, and the current system is ignorant of problems caused by application faults, such as immediate hardware control, kernel code execution, and kernel data corruption, so the system may collect incorrect data, or reduce the node availability. As we cannot write safe applications all the time and sensor node hardware does not easily detect application errors, techniques to ensure system resilience at the operating system level should be developed.

Currently available operating systems for wireless sensor networks include TinyOS [3], MANTIS [4], and SOS [5]. The component-based and event-driven TinyOS produces a single code image where the kernel and application are statically linked. There is no distinction between kernel and application, so a badly written application can cause the system to fail [6]. MANTIS provides a multithreaded programming model, but it is not free from the possibility of user errors due to a

statically linked image, as is the case of TinyOS. SOS separates the kernel and application modules via dynamically loadable modules. This technique, however, does not include measures to restrict the application from accessing kernel data or other application data, and calling kernel code abnormally. Concerning the system errors, some operating systems use a watchdog timer, but it is not easy to recognize and handle problems such as memory access beyond the application area, immediate hardware control, and error repetition. Users have to reset sensor nodes directly to recover from specific errors [7]. Meanwhile, Maté [6] is a virtual machine for wireless sensor networks. The interpreter in the virtual machine enables the detection of hazardous instructions in the runtime, but there are also limitations, such as performance degradation or additional efforts to learn new programming languages for the virtual machine [5].

The operating system mechanism for error-free wireless sensor networks should operate with software assistance. The mechanism ought to protect the kernel from applications and support diverse applications in a multithreaded environment. However, previous work on the software approach to ensure system safety has typically focused on the MMU-equipped general purpose systems, and on the single-threaded model. SFI (Software Fault Isolation) [8] modifies the data access via indirect addressing or jump instructions to be executed in a single allocated segment, which requires MMU. SFI was designed for RISC architectures that have fixed-length instructions. [12] suggests that SFI can cause illegal jump operations when it is implemented on variable-length instruction architecture, which are usually adopted in sensor node platforms. Proof-carrying Code [9] evaluates the application using a safety policy in compile time. However, as the automatic policy generator does not exist, the technique cannot be applied directly to real systems. Programming language approaches include Cyclone [10], Control-C [11], and Cuckoo [12]. All of these are based on the C programming language, but users should be aware of the different usages of pointers and arrays. Cyclone requires hardware supports for stack safety; Control-C aims to offer system safety without runtime checking, although additional hardware is required for stack safety and the language does not guarantee fault-

free array indexing; Cuckoo provides system safety without hardware supports. The overhead of Cuckoo is, however, not trivial – being almost double the size of the optimized GCC execution time.

This paper presents a resilient operating system mechanism for wireless sensor networks. The proposed mechanism is designed to apply to the common sensor node platform with RETOS, the preemptively multi-threaded operating system we are currently developing, and enables sensor node systems to run safely from errant applications without hardware supports. The mechanism implemented in the RETOS kernel detects harmful attempts on system safety by applications, and terminates the badly-written application programs appropriately. The effectiveness of the proposed mechanism is validated by experiments conducted on a commercial sensor node device running the RETOS operating system.

The rest of this paper is organized as follows: Section 2 describes background on the system software and the hardware platform used in the paper; Section 3 explains the proposed safety mechanism; Section 4 validates the effectiveness of the mechanism via real experiments; and Section 5 concludes the paper.

2. RETOS Architecture

The proposed safety mechanism is based on the RETOS operating system. Figure 1 illustrates the system organization. RETOS provides preemptive multithreading based on a dual mode operation that cooperates with static/dynamic code checking to protect the system from application errors. (Dual mode operation normally refers to running with two hardware privilege levels, but in this paper, we use the terminology as the kernel/user separated operation.) To elevate the memory efficiency of dual mode operation, a single kernel stack is maintained in the RETOS kernel. This technique restricts thread preemption to be performed in user mode, but the amount of required system memory is decreased. With the thread preemption, hardware contexts are saved in each thread's TCB (Thread Control Block) due to kernel stack sharing. The mechanism for ensuring system safety with dual mode operation and static/dynamic code checking is described in Section 3 in detail.

RETOS supports the POSIX 1003.1b real-time scheduling interface. Threads are scheduled by three scheduling policies: SCHED_RR, SCHED_FIFO, and SCHED_OTHER. Thread scheduling is done in kernel level, while synchronization methods reside in user level. When synchronization primitives like *mutex* are executed, the thread library disables interrupts in user level and tries to acquire the resource. On an unsuccessful case, the library inserts the thread information to the resource waiting list and blocks the thread. After the resource is unlocked, the library sequentially wakes up the threads in

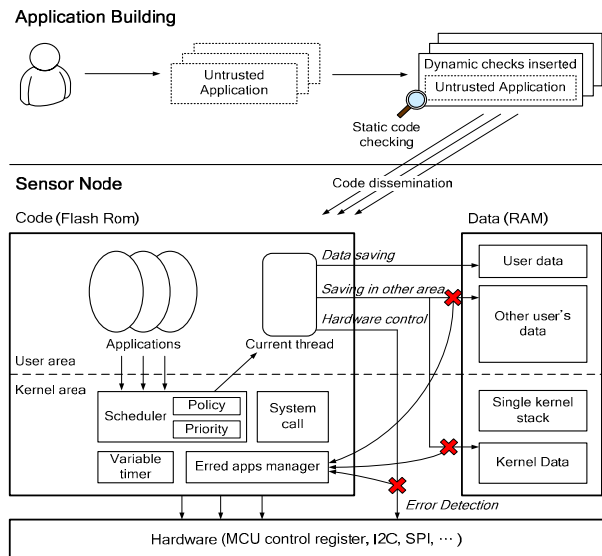


Figure 1. RETOS System Overview

the waiting list.

In the RETOS system, applications are separated from the kernel code, and several applications can be loaded and executed dynamically. To exploit runtime application loading in a single address space, the address relocation technique is employed. The method produces an address relocation table during the compile time, and relocates the addresses of the binary to the ROM/RAM address, which is allocated for the application. The memory manager for the data/stack area management of loadable applications is based on first-fit allocation. Dynamic memory allocation is currently available only for the kernel codes.

RETOS is being implemented on the Tmote Sky [13]. The mote is based on the TI MSP430 F1611 (8Mhz, 10Kb internal RAM, 48Kb internal Flash) microcontroller, and the Chipcon 2420 RF module. The MSP430 instruction set consists of core instructions and emulated instructions. There are three types of core instructions: dual-operand, single-operand, and jump. Users program sensor applications with the standard C and the Pthread library.

3. Ensuring System Safety

The proposed mechanism ensures system resilience via dual mode operation and application code checking. Since general microcontrollers such as MSP430 only have a single address space, the kernel and applications exist in the same address space. Dual mode operation logically separates the kernel and the user execution area. Mote devices cannot protect applications' address spaces or keep the hardware resources controlled. Application code checking which consists of both static and dynamic techniques solves the problem. The concept of the safe

operating system mechanism proposed in the paper is summarized in Figure 1. User applications become trusted code through static/dynamic code checking. Applications loaded on the system are allowed to store data and to execute codes in their own resources, but application errors that were not detected at the compile time are reported to the kernel. When the errors are reported, the kernel informs users of the illegal instruction address and safely terminates the program.

3.1. Dual Mode Operation

Dual mode separates the kernel from the user execution area to perform application code checking in the target operating system, RETOS. Since the static/dynamic code checking evaluates if the application modifies data or issues codes in its allocated area, preemption in the system which executes kernel and user code in the same stack would invoke problems. For example, a thread which has access rights to other threads in a common application group would destroy the stored kernel data, such as a return address and hardware contexts, in the blocked thread stacks.

In the proposed mechanism, dual mode is operated by stack switching. Applications in the user mode use the user stack, and the stack is changed to the kernel stack for system calls and interrupts handling. Figure 2 shows the dual mode operation for system call handling on the proposed system. System calls are implemented by a function call on the TI MSP430 microcontroller, so the return address remains in the user stack, thereby leaving it to be modified by other threads. Upon system call, the current stack pointer indicating the user stack and the return address are stored in the PCB, and the runtime stack is changed to the kernel stack. Therefore, the return address validation is necessary before returning to the user mode. The case of interrupt handling is similar to the one illustrated in Figure 2. When an interrupt is invoked, MSP430 pushes the program counter in the current stack and jumps to the corresponding interrupt handler. The handler function switches to the kernel stack, if the system was in the user mode, and checks the return address after processing.

Dual mode may incur memory overhead on resource-constraint sensor nodes due to the per-thread kernel stack. To save memory usage, the proposed mechanism maintains a single kernel stack in the system. Kernel stack sharing means that the system cannot arbitrarily interleave execution flow, including thread preemption, while they are in the kernel mode. Instead, the kernel provides the deferred invocation. System calls, such as radio communication, register their long-running tasks to be executed later and return as soon as possible. Thread switching is performed right before returning to user mode, that is, the time when all work pushed on the

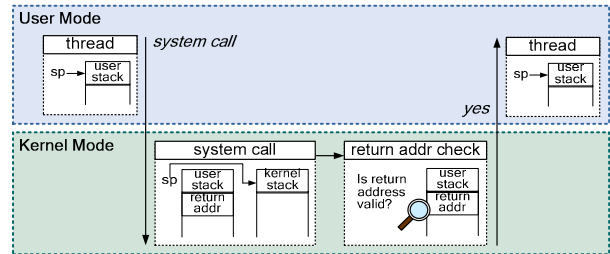


Figure 2. Dual Mode Operation

kernel stack is finished. Although the single kernel stack is unable to preempt threads in the kernel mode, it enables the memory efficient implementation of dual mode operation in the soft real-time system, where the preemptive kernel is not strongly required. In addition to memory overhead, mode switching overhead is found in interrupts and system calls handling. Section 4 evaluates such overhead.

3.2. Static/Dynamic Code Checking

Static/dynamic code checking sets restrictions on an application for using data and the code area within the application itself, and restricts direct hardware resource manipulation. The proposed technique inspects the destination field of machine instructions. The destination address evaluation prevents the application from writing or jumping to an address outside its logically separated portion. The destination field is observed if it is a hardware control register, such as the MCU status control register. The source field of instructions can also be examined to prevent the application from reading kernel or other applications data. We, however, do not adopt the mechanism because of the security issues as well as computation overhead. The code checking technique considers non-operand instructions such as *ret* and *eint/dint*. This technique evaluates a return address of function for *ret*, and looks up *eint/dint* to disallow immediate hardware control.

The technique consists of static and dynamic code checking. Direct/immediate addressing instructions, pc-relative jumps and *eint/dint* are verified during the compile time. As we assume the library codes are safe, the libraries are not checked in our implementation. Indirect addressing instructions are verified in runtime due to unpredictable destination addresses. Runtime checking is required of the *ret* instruction, because the return address can be affected by buffer overrun. Figure 3 shows the application building sequence, including static/dynamic code checking. Every source code of the application is compiled to assembly code; then checking code is inserted to the place where the dynamic code checking is required. After dynamic code insertion, a



Figure 3. Application Building Sequence

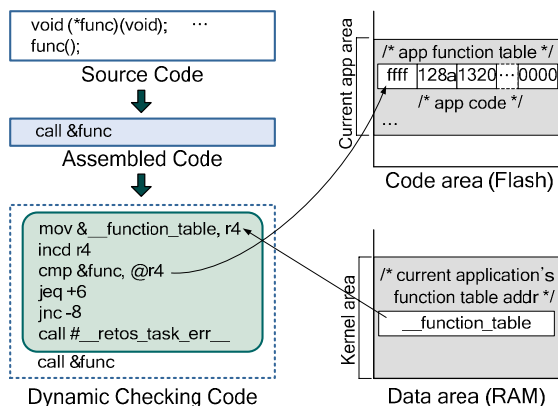


Figure 4. Dynamic Code Checking

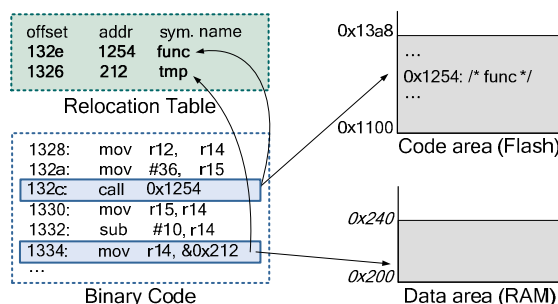


Figure 5. Static Code Checking

binary image is created via compiling and linking, and the static code checking is then performed on the binary.

Figure 4 is the example of dynamic code checking for TI MSP430. Since the technique for indirect calls inspects the destination address using the application's function table, it makes the instruction unable to branch directly to a hazardous instruction by passing the inserted checking code or to the midst of the instruction. Here, an application binary should include its function table in the header, and the kernel should provide a variable to save the address of the current application's function table and update it at the thread scheduling. The technique for call/push instructions does not consider stack overflow because the stack usage verification is conducted at the function prologue by way of the stack depth count in the function. The indirect store instruction is examined similarly as shown in Figure 4. The difference is to check every dynamically allocated RAM area for threads in the application using the linked list. Note that the r4 register shown in the Figure 4 checking code is configured to not

be used by the MSP430-GCC.

Figure 5 shows an example of static code checking. This technique identifies the existence of the destination address in the relocation table, which has all the symbol information in the source code, and the offsets of those instructions referring to the symbols. If the address is not in the relocation table, the instruction will be confirmed as incorrect. For instance, the destination address of the instruction call 0x1254 in Figure 5 exists in the relocation table, and its offset is identical to the one in the table. In addition, the technique estimates the destination address within a dedicated area because data and code size is limited in the mote system. The address 0x1254 in the figure is between the application start address 0x1100 and the end address 0x13a8. Hence, the instruction call 0x1254 is proved correct. The *mov* instruction in the figure is also checked to be correct by the same method.

4. Evaluation

This section describes the experiment results of the proposed resilient operating system mechanism, and analyzes its performance characteristics. Both the mechanism and RETOS have been implemented for the TI MSP430 F1611 (8Mhz, 10Kb RAM, 48Kb Flash) based Tmote Sky hardware platform.

4.1. Functionality Test

To adequately evaluate the error management of the proposed mechanism, we classify the safety domain of applications into four parts: stack, data, code and hardware. Stack safety means the prevention of stack overflow due to function calls and local variable handling. Data safety implies the protection of the kernel and other applications data against illegal access, and code safety restricts the execution of the kernel and other applications code. Hardware safety implies whether the system can be protected from immediate hardware control. Table 1 shows the examples of the codes for each safety domain. A recursive call verifies the stack safety of the proposed mechanism. Modifying data outside the application's area, by using a directly addressed pointer and deviated array indexing, is done to analyze data safety. A directly addressed function call and a corrupted return address due to buffer overflow are used to test code safety. An interrupt disabling code and a Flash ROM writing code are used for the hardware safety check.

Following the functionality tests, the proposed mechanism is proved to ensure system resilience against hazardous application codes. An application, including a recursive call, an illegal array indexing, and a buffer overflow, is reported to the kernel by dynamic code checking and is terminated safely. The static check detects storing at and calling the address outside the

Table 1. Example Codes for Functionality Test

	Test Set
Stack safety	- General/Mutual recursive call void foo() { foo(); }
Data safety	- Directly addressed pointer int *tmp = 0x400; *tmp = 1; - Array indexing int array[10]; /* array in heap area */ for(i = 10; i > 0; i--) array[i-100] = i;
Code safety	- Directly addressed function pointer void (*func)(void) = 0x1000; func(); - Buffer overrun (damaging return address) void func() { int array[5], i; for(i = 0; i < 10; i++) array[i] = 0; }
Hardware safety	- Disable interrupt asm volatile ("dint"); - Flash rom writing (memory mapped regs.) FCTL1 = FWKEY+WRT;

Table 2. Dual Mode Overhead (cycle)

	Single mode	Dual mode
system call (led toggle)	264	302
system call (radio packet send)	352	384
timer interrupt (invoked in kernel)	728	728
timer interrupt (invoked in user)		760

application, which are caused by directly addressed pointers, interrupt disabling and Flash ROM writing. In order to compare the existing sensor network operating systems, that provide no safety mechanism, TinyOS v1.1.13 is selected to execute the application code in Table 1. The application using the directly addressed pointer and deviated array indexing does not perform well. The codes for recursive call, directly addressed function pointer, buffer overflow, interrupt disabling and the Flash ROM writing crash the system. The watchdog timer reboots the system sometimes, but the system is not restored most of the time.

4.2. Overhead Analysis

The proposed mechanism may have some overhead due to dual mode operation and dynamic code checking. The first set of experiments aims to analyze the performance of dual mode operation. We have implemented two versions of RETOS, dual mode and single mode, to measure performance degradation from mode switching and the return address check. Table 2 shows the results.

The experiment data shown in Table 2 denotes approximately 32~38 cycles of computational overhead for system calls, toggle a led and send a radio packet, and dual mode operation. At the timer interrupt handling, however, operation time differs from the interrupt that occurred in kernel mode and user mode. As the stack is not changed and the return address checking is omitted in kernel mode, the result of handling the timer interrupt invoked in kernel mode on the dual mode system is identical with the result on single mode. The overhead

Table 3. Dynamic Code Checking Overhead (cycle)

	No check	Dynamic check	Overhead(%)
MPT_backbone	40326	40508	0.5
MPT_mobile	386566	394624	2.1
R_send	24815	26176	5.5
R_recv	4704	5010	6.5
Sensing	1056	1096	3.8
Pingpong	1243	1347	8.4
Surge	58723	62688	6.7

Table 4. Application Code Size Comparison (bytes)

	No check	Dynamic check	Overhead(%)
MPT_backbone	614	682	11.1
MPT_mobile	8726	10046	15.1
R_send	946	1014	7.2
R_recv	902	1004	11.3
Sensing	774	835	7.9
Pingpong	478	510	6.7
Surge	1436	1610	12.1

of the timer interrupt invoked in user mode on the dual mode system is 32 cycles, which is similar to the case of system calls.

The second set of experiments was conducted to observe the execution time overhead of dynamic code checking. We compare the codes with dynamic checks to original applications running RETOS by calculating average instruction cycles per second during 10 minutes. To measure the overhead of inserted codes, we consider the execution time running in user mode. Seven sensor network applications are used for the test. *MPT_mobile* and *MPT_backbone* are decentralized multiple object tracking programs [14]. When *MPT_mobile* node moves around, it sends both an ultrasound signal and a beacon messages every 300ms to nearby *MPT_backbone* nodes. *MPT_backbone* nodes report their distance to the mobile node, and *MPT_mobile* computes its location using trilateration. *R_send* and *R_recv* are programs to send and receive radio packets with reliability. *Sensing* samples the data and forwards it to the neighbor node. *Pingpong* makes two nodes blink in turns by means of the counter exchange. *Surge* is a multihop data collecting application which manages a neighbor table and routes the packet.

Table 3 shows that applications using dynamic code checking have 0.5~8.4% performance degradation. *MPT_mobile*, which requires the longest processing time, generated 2% more overhead when the protection mechanism was used; the amount of calculation time caused by non-hardware multiplier is much larger than dynamic checking. Whereas, *R_recv*, *Pingpong*, and *Surge*, all of which require more memory access than complex arithmetic calculations, shows larger overhead. Since the dynamic code checking requires code insertion, the technique increases the application code size when compared to the original one. Table 4 shows the code size comparison. The increases are 4~12%. However, the application is inherently small, being separated from

kernel, and hence code size increases are not considerable for the internal Flash ROM of TI MSP430 or AVR ATmega128L, the common microcontrollers used for sensor node devices.

The performance of dual mode operation, the computational overhead and code size increment of dynamic code checking are analyzed. Since the system stays idle most of the time due to sensor network application characteristics, energy consumption for the proposed mechanism may be almost the same as existing systems. The overhead of dynamic code checking, however, heavily depends upon application types and the programmers' coding style. Generally, frequent usages of local variables, function calls, and pointers cause more overhead.

5. Conclusion

In this paper, we presented a resilient operating system mechanism for wireless sensor networks, based on dual mode operation and static/dynamic code checking. The proposed mechanism guarantees stack, data, code and hardware safety on MMU-less hardware without restriction of the standard C language. Dynamic code checking with dual mode operation is reported in approximately 8% of execution time overhead on the TI MSP430 processor.

The experiments were conducted under the assumption that the libraries always operate safely, and the code checking techniques do not inspect library codes. In real situations, however, if a user passes an invalid address to `memcpy()` then the function would destroy memory contents. For a solution, we considered recompiling standard libraries with our techniques or making wrapper functions that check address parameters. Also, our mechanism cannot handle the case where a user intentionally skips the code checking sequences or modifies a program binary. To prevent malicious usage, user authentication on code updating would be required.

RETOS, safety mechanism applied operating system, is currently being developed by our research group. Although this paper shows that RETOS protects the system from errant applications, supplement for libraries and system calls is required in order to program applications easily. We are presently developing a network stack for energy efficient radio communication on RETOS, as well as implementing device drivers for diverse sensors and porting to other processors.

Acknowledgements

This work was supported in part by the National Research Laboratory(NRL) program of the Korea Science and Engineering Foundation (2005-01352), and the ITRC programs(MMRC) of IITA, Korea.

References

- [1] D. Culler, P. Dutta, C. T. Ee, R. Fonseca, J. Hui, P. Levis, J. Polastre, S. Shenker, I. Stoica, G. Tolle, and J. Zhao, "Towards a Sensor Network Architecture: Lowering the Waistline," *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, USA, June 2005.
- [2] V. Handziski, J. Polastre, J. Hauer, C. Sharp, A. Wolisz and D. Culler, "Flexible Hardware Abstraction for Wireless Sensor Networks", *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN'05)*, Istanbul, Turkey, January 2005.
- [3] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for network sensors," *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, Cambridge, MA, USA, November 2000.
- [4] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms," *ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks*, vol. 10, no. 4, August 2005.
- [5] C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava, "SOS: A dynamic operating system for sensor networks," *Proceedings of the 3rd International Conference on Mobile Systems, Applications, And Services (MobiSys'05)*, Seattle, WA, USA, June 2005.
- [6] P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, San Jose, CA, USA, October 2002.
- [7] Deluge: TinyOS Network Programming, <http://www.cs.berkeley.edu/~jwhui/research/deluge/>
- [8] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Software-based fault isolation," *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP'93)*, Asheville, NC, USA, December 1993.
- [9] G. C. Necula, "Proof-carrying code," *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, Paris, France, January 1997.
- [10] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, CA, USA, June 2002.
- [11] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory safety without runtime checks or garbage collection," *Proceedings of Languages, Compilers and Tools for Embedded Systems (LCTES'03)*, San Diego, CA, June 2003.
- [12] R. West, and G. T. Wong, "Cuckoo: a Language for Implementing Memory and Thread-safe System Service", *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC'05)*, Las Vegas, NV, USA, June 2005.
- [13] Moteiv, Inc., <http://www.moteiv.com>.
- [14] W. Jung, S. Shin, S. Choi, and H. Cha, "Reducing Congestion in Real-Time Multi-Party Tracking Sensor Network Application," *Proceedings of the 1st International Workshop on RFID and Ubiquitous Sensor Networks*, Nagasaki, Japan, December 2005.

Transparent Contribution of Memory

James Cipar Mark D. Corner Emery D. Berger*

*Department of Computer Science
University of Massachusetts-Amherst, Amherst, MA 01003
{jcipar, mcorner, emery}@cs.umass.edu*

Abstract

A multitude of research and commercial projects have proposed *contributory* systems that utilize wasted CPU cycles, idle memory and free disk space found on end-user machines. These applications include distributed computation such as signal processing and protein folding, peer-to-peer backup, and large-scale distributed storage. While users are generally willing to give up unused CPU cycles, the use of memory by contributory applications deters participation in such systems. Contributory applications pollute the machine's memory, forcing user pages to be evicted to disk. This paging can disrupt user activity for seconds or even minutes.

In this paper, we describe the design and implementation of an operating system mechanism to support transparent contribution of memory. A *transparent memory manager* (TMM) controls memory usage by contributory applications, ensuring that users will not notice an increase in the miss rate of their applications. TMM is able to protect user pages such that page miss overhead is limited to 1.7%, while donating hundreds of megabytes of memory.

1 Introduction

A host of recent advances in connectivity, software, and hardware has given rise to *contributory systems* for donating unused resources to collections of cooperating hosts. The most prominent examples of deployed systems of this type are Folding@home and SETI@home, that donate excess CPU cycles to science. Other examples include peer-to-peer file sharing applications that enable users to donate outgoing bandwidth and storage and receive bandwidth and storage in return. The research community has been even more ambitious, proposing

systems that harness idle disk space to provide large-scale distributed storage [8, 4, 3]. All of these applications, be they primarily processing or storage, require a donation of idle memory, since contributory applications consume memory for mapped files, heap and stack, as well as the buffer cache.

Users are only willing to participate in such systems if contribution is transparent to the performance of their ordinary activities. Unfortunately, contributory applications are not at all transparent, leading to significant barriers to widespread participation. Contributing processing and storage leads to memory pollution, forcing the eviction of the user's pages to disk. The result is that users who leave their machines for a period of time can be forced to wait for seconds or even minutes while their applications and buffer cache are brought back into physical memory. In this paper, we show that this figure can grow to as high as 50% degradation after only a five minute break.

A number of traditional scheduling techniques and policies, such as proportional shares [10], can prevent only some kinds of interference from contributory services. For instance, if the owner is not actively using the machine, a contributory service can use all of the resources of the machine. When the user resumes work, the resources are reallocated to give fewer resources to the contributory service. However, while the resource allocation of a network link or processor can be changed in microseconds, faster than any user can notice, memory allocation does not work well with the same strategies. Due to the reliance on relatively slow disks, the memory manager can take minutes to page while the user waits for an unacceptable amount of time.

As a solution to this problem we present the *transparent memory manager* (TMM), which protects *opaque* applications from interference by *transparent* applications. *Opaque* applications are those for which the user is concerned with performance. Typically these will be all of the applications run for the user's own benefit on

*This material is based upon work supported by the National Science Foundation under CAREER Awards CNS-0447877 and CNS-0347339. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

their workstation. Such opaque applications may be interactive, batch, or even background applications, but users prioritize opaque use over contributory applications. *Transparent* applications are those which the user is running to contribute to a shared pool of resources. TMM ensures that contributed memory is transparent: opaque performance is identical whether or not the user contributes memory.

A key feature is that TMM is dynamic: it automatically adjusts the allocation given to opaque and transparent applications. This is in sharp contrast with schemes like Resource Containers that statically partition resources [5]. Such static allocations suffer from two problems. First, a static allocation will typically waste resources due to the lack of statistical multiplexing. If one class is not using the resource, the other class should be able to use it, and static allocations prevent this. Second, it is unclear how users should choose an appropriate static allocation—as the workload changes, the best allocation changes drastically. Third, the primary concern of users is the effect that running a service has on the performance of their computers. It is not possible for a user to determine how that translates into an allocation.

Another important property of TMM is that it is a global policy. It considers the behavior of all opaque applications together when determining the limits, and limits to total memory use consumed by all transparent applications. System calls such as `madvise` can be used by individual applications to limit their own impact of system performance, but many transparent applications running concurrently would have to coordinate their use of `madvise` to ensure that their aggregate memory use did not exceed the limit. Furthermore, the necessary modification of application code is contrary to our design goals.

The rest of the paper describes the design in Section 2, implementation in Section 3, and evaluation in Section 4. Our evaluation shows that TMM is able to limit the effect of transparent applications to a 1.7% increase in page access times while allocating hundreds of megabytes to transparent applications. An expanded version of this work, including a mechanism for transparent storage, is available in a technical report [2].

2 Design

The goal of Transparent Memory Management (TMM) is to balance memory allocations between classes of applications. TMM allocates as much memory as it can to contributory applications, as long as that allocation is transparent to opaque applications. The insight is that opaque processes are often not using all of their pages profitably and can afford to donate some of them to contributory applications. The key is to decide how many pages to donate and to donate them without interfering

with opaque applications.

Here we provide an overview of TMM: (i) it samples page accesses with a lightweight method using page reference bits, (ii) it determines allocations using an approximate Least Recently Used (LRU) histogram of memory accesses based on the sampled references, (iii) it evicts opaque pages in an approximate LRU manner to free memory for transparent applications, (iv) it ages the histogram to account for changing workloads, and (v) it filters out noise in page access to keep the histogram constant even when the user takes a break. We omit the details of (i) and describe the rest in more detail below.

2.1 Determining Allocations

The key metric in memory allocation is the access time of virtual memory. As TMM donates memory to transparent tasks, opaque memory will be paged out, increasing the access time for opaque pages. Thus TMM determines the amount of memory to contribute by calculating what that allocation will do to opaque page access times. The mean access time (MAT) for a memory page is determined from the miss ratio (μ), the time to service a miss (m), and the time to service a hit (h):

$$\text{MAT} = \mu \cdot m + (1 - \mu) \cdot h. \quad (1)$$

If the system allows background processes to use opaque pages, it will increase the miss rate μ of opaque applications by a factor of β , yielding an increase in MAT by a factor of:

$$\frac{\text{MAT}'}{\text{MAT}} = \frac{\mu \cdot \beta \cdot m + (1 - \mu \cdot \beta) \cdot h}{\mu \cdot m + (1 - \mu) \cdot h}. \quad (2)$$

In the case of a page miss, the page must be fetched from the page's backing store (either in the file system or from the virtual memory swap area), which takes a few milliseconds. On the other hand, a page hit is a simple memory access, and takes on average just a few tens of nanoseconds. Because these factors differ by many orders of magnitude, TMM can estimate that the average page access time is directly proportional to the number of page misses. Increasing the miss rate by β will make the ratio in (2) approximately β . This ratio is valid as long as there are some misses in the system. TMM uses a value of $\beta = 1.05$ by default, assuming that most users will not notice a 5% degradation in page access times.

2.2 LRU Histogram

Limiting an allocation's effect to the factor β requires knowing the relationship between memory allocations and the miss rate. This relationship can be directly determined using a Least Recently Used (LRU) histogram,

also known as a page-recency graph [9, 11, 12] or a stack distance histogram [1]. An LRU histogram allows TMM to estimate which pages will be in and out of core for any memory size. Using this histogram, TMM can determine what the miss rate for opaque processes would be for any allocation of pages to transparent processes. Building the histogram depends on sampling page references, and we use a method based on reference bits [2]. Previous research shows that these sampling and approximation-based approaches work well even as many operating systems use approximations of LRU, such as CLOCK [12], or 2Q [7].

In an LRU histogram H , the value at position x represents the number of accesses to position x of the queue. Thus $\sum_{i=0}^x [H(x)]$ is the number of accesses to all positions of the histogram up to and including x . This value is approximately equal to the number of page hits that would have occurred in a system that had a memory size of x pages. Subtracting this value from the total number of accesses in the workload gives the number of misses for that memory size. It is important to note that the LRU histogram contains all of the *virtual* pages in the system. With only the physical pages, it would not be able to predict what the miss rate would be given more memory pages. A sample cumulative histogram and memory allocation is shown in Figure 1.

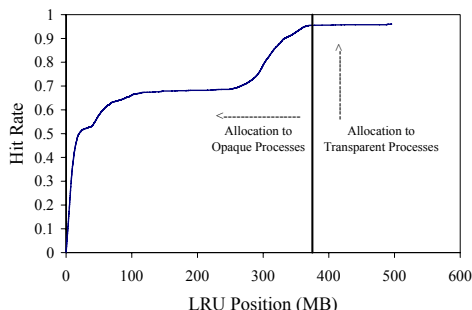


Figure 1: This figure demonstrates a sample histogram and allocation. In this case, the user has a working set size of approximately 390MB and can afford to donate the rest of the physical memory.

2.3 Page Eviction

When applications allocate pages, the operating system will first determine if there are free pages. If there are, it simply hands them to the process, regardless of the limits determined from the histograms—there is no reason to deny use of free pages. However, when free pages run low, the OS will force the eviction of other pages from the system and must choose between transparent and opaque pages. The choice depends on the limits de-

termined by the factor β , and the current allocation of pages in the system. If both opaque and transparent applications are above their limits, or both are below their limits, it favors opaque applications and evicts transparent pages. Otherwise it will evict from whichever class is above its limit.

2.4 Aging the Histogram

We age the histogram over time using an exponentially weighted moving average. TMM keeps two histograms: a permanent histogram that TMM computes limits from, and a temporary histogram that records only the most recent activity. After some amount of time t , we first divide the temporary histogram by the total number of accesses which happened in that aging period so that the values represent hit to miss ratios. We then add the temporary data into the permanent histogram using an exponential weighted moving average function and then recompute the cumulative histogram. We adopt a common value $\alpha = \frac{1}{16}$ and adjust the time t to match this.

The difficult part lies in tuning t . If TMM is not agile enough (too stable), rapid increases in opaque working set sizes will not be captured by the histogram, transparent applications will be allocated too many pages, and opaque page access times will suffer. If TMM is not stable enough (too agile), infrequently-used opaque applications will not register in the histogram and may be paged out.

To deal with this, we adopt a policy that is robust but favors opaque pages. Normally, t is set to 10 minutes for stability. This value is large enough that applications used in the last day or two bear enough significance in the histogram to force the memory limit to not page them out. However, to remain agile, the system must move the limit in response to unusually high miss rates. If TMM notices that the page misses have violated the stated goal, it adopts a more agile approach, using the most recent sample as the temporary histogram and immediately averaging it with the stable one. This policy has the disadvantage of stealing pages from transparent applications based on transient opaque use, but it is required to favor opaque applications over transparent ones.

2.5 Dealing with Noise

The goal in aging the histogram is to detect phase shifts in user activity over time. When the user is not actively using the machine, the histogram should remain static, directing TMM to preserve the opaque pages in the cache. However, we have observed that even when not actively using the machine, our Linux installation still incurs many page references from opaque applications. If left long enough, TMM misinterprets these accesses as

shifts in user behavior and will change the allocation of pages to opaque applications. As there are only a small number of these pages, when the user is not using the system, it appears that the working set has very high locality. TMM will act improperly, allowing transparent allocations to consume a large number of pages in the system.

To avoid this, TMM filters its page samples. To decide whether or not to age the histogram after an aging period of t seconds, we only consider the opaque page accesses that have touched the LRU queue at a point farther than 10% of the total. If there were more than ten such accesses per second, we age the histogram. This policy also implies that we need to guarantee a minimum of 10% of the system's memory to opaque application, a reasonable assumption. We found that the idle opaque activity rarely touches pages beyond the 10% threshold. We observed that with filtering, TMM never ages the histogram during periods of disuse.

3 Implementation

TMM requires a number of kernel modifications as well as several user-space tools¹. Inside the kernel, we implemented tracing methods that periodically mark pages as unreferenced, and then later test them for MMU-marked references. The list of page references and recent evictions are passed through a `/dev` interface. A user space tool written in C++ reads these values and tracks the LRU queue. This tool computes the transparent and opaque limits and passes the limits back into the kernel through a `/sys` interface.

Additionally, we have augmented the Linux eviction policy kernel with our LRU-directed policy which enforces memory limits. Another user-space tool, `maketransparent`, allows users to mark processes as transparent. All pages that the transparent processes use are then limited using TMM. Pages that are shared between opaque and transparent pages are marked as opaque pages, but to ignore noise caused by transparent process access, hits to those pages are not traced.

4 Evaluation

In evaluating TMM, we sought to answer the following questions: (i) How well does TMM prevent transparent processes from paging opaque memory out, and how does TMM's dynamic technique compare to static allocation? (ii) What is the transient performance of TMM? (iii) What is the overhead in using TMM?

The primary function of TMM is to ensure that contributory processes that use memory do not interfere with the user's applications. To show TMM's benefits, we

simulate typical user behavior: we use several applications, take a coffee break for five minutes, and return to using similar applications. During the coffee break, the machines runs a contributory application. As a contributory application, we use a program called POV-Ray, a widely-used distributed rendering application. The rendering benchmark we used with POV-Ray caused it to have a working set size greater than the physical memory on our test platform. Most contributory applications do not use memory this aggressively; POV-Ray could be considered a "worst-case" contributory application. For this experiment, we compare five systems with three different sets of opaque applications. The five systems are as follows: vanilla Linux, TMM with three different static allocations for opaque applications (25%, 50%, 75% of the physical memory), and TMM using its histogram-based limiting method. For vanilla Linux, we present results with and without the contributory application. The three different sets of applications represent different user activities with different working set sizes: Small (Mozilla), Medium (Mozilla, OpenOffice, and KView), and Large (Gnuplot with very large data set). We track the average and maximum number of page misses per second recorded in the first minute after the user comes back from break and present the results in Figures 2 and 3. Note that we rebooted between each trial, and we only monitored page misses incurred by opaque processes.

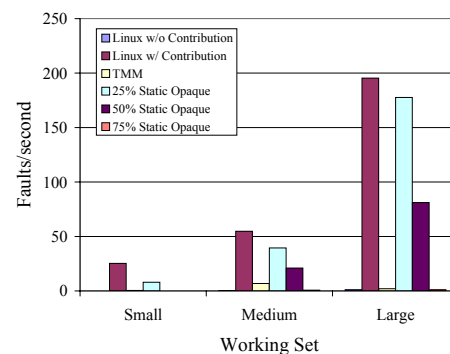


Figure 2: This figure shows the average page faults/sec in the first minute after resuming work. TMM performs much better than an unmodified system, and better than static limits, except for a very high static limit.

The first thing to note is that with a vanilla Linux kernel, the system running the contributory application performs very poorly, incurring as many as 190 page faults/sec on average. Assuming that the application is page fault limited, and given an average miss latency of 2.5ms, these faults cause a 50% slow down in the execution of the application. Qualitatively, we have observed that this is highly disruptive to the user. Second,

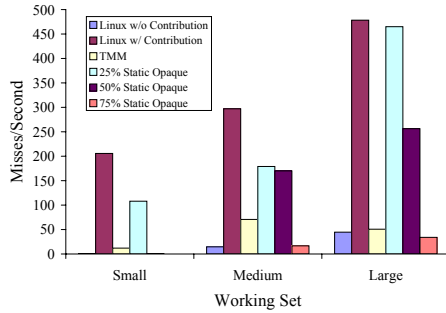


Figure 3: This figure shows the maximum page faults/sec in the first minute after resuming work. Short term violations of the target 5% slowdown are possible, but TMM performs better than unmodified and static systems.

for each static limit, there is a workload that performs worse than TMM, except for the 75% limit. In that case, the static limit performs well. However, we could have easily constructed a working set size larger than 75%, and TMM would have produced far fewer page misses than the static allocations.

Most importantly, the performance of TMM is comparable to the performance of the vanilla Linux system without contribution. The largest number of average page faults that TMM incurs is for the medium size working set, at 6.8 page faults/sec. By the same calculation used above, this faulting rate causes a 1.7% slow down in the opaque applications, well within our goal of 5%. This demonstrates TMM's ability to donate memory transparently. The few pages that TMM does evict could be recovered by making Linux's page eviction more LRU-like. As shown in Figure 3, short-term violations of our goal are possible—TMM statistically guarantees average performance over long periods. Nonetheless, TMM provides better short-term performance than the static limits (except 75%), and much better performance than unmodified Linux. For the large working set, even Linux without contribution does incur some page faults.

To measure the actual amount of memory that TMM donates to transparent applications, we ran the interactive phase of our benchmarks, waited for the limit to stabilize, and recorded the transparent limits. We also recorded the amount of memory donated under the static limits. The static limits apply to the limit on opaque memory and thus transparent memory contribution also depends on consumption by the operating system, thus a single static limit donates slightly different amounts of memory under different workloads. The results are presented in Figure 4.

When comparing the amount of memory donated by each system, we show that TMM is conservative in the

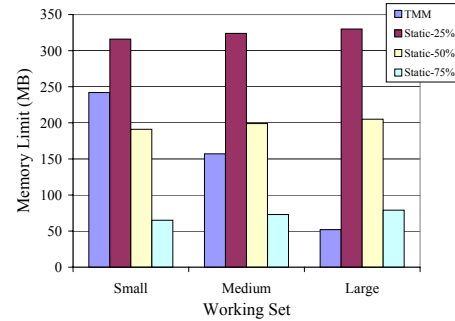


Figure 4: This figure shows the amount of memory in MB donated to transparent processes.

amount of memory it donates, favoring opaque page performance over producing a tight limit. Nonetheless, by considering both Figures 2 and 4, the results show that there is a working set for which the static limits fail to both preserve performance and contribute the maximum amount of memory. TMM succeeds in achieving both of these goals for every working set.

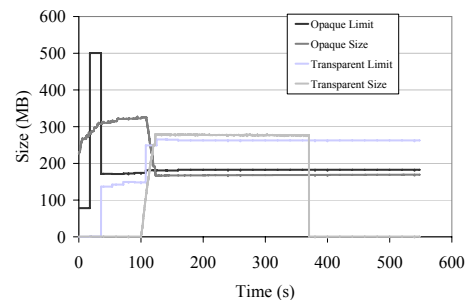


Figure 5: This figure shows a sample timeline of limits and utilization of the TMM system.

The next experiment demonstrates how TMM behaves over time. We conduct an experiment similar to the previous one and graph the memory use and memory limits that TMM sets. A timeline is shown in Figure 5. At the beginning of the timeline, the set of opaque processes is using 320 MB of memory and there are no transparent applications in the system. TMM has set a limit for both opaque and transparent processes, but as there is no memory pressure in the system, the memory manager lets opaque processes use more memory than the limit. Note that at this time, the sum of the transparent and opaque limits is far less than the physical memory of the machine (512 MB), the rest of the memory is in free pages. At 30 seconds we start a transparent process that quickly consumes a large amount of the free memory. TMM now sees a larger pool to divide and increases the transparent limit. TMM does not adjust the opaque limit as the user has not changed behavior and does not

need any more memory than the limit allows. The transparent process is now causing memory pressure in the system, forcing pages out. The number of opaque pages is over its limit so it loses them to the evictor. The graph exhibits some steady state error. We have tracked this to the zoned memory system that Linux uses, something we will correct in a redesigned evictor.

Lastly, it is important to note the CPU and memory overhead of TMM. TMM consumes approximately 6% of the CPU during startup and less than 2% once it has established an accurate LRU queue. This CPU time is primarily due to running TMM in user space requiring many user-kernel crossings to exchange page references and limits. TMM uses approximately 64MB of memory. This large memory overhead is due to duplicating the kernel's LRU list in user-space, another straightforward optimization. A kernel implementation of TMM will reduce both of these overheads significantly.

5 Related Work

The CacheCOW system [6] generally defines the problem of providing QoS in a buffer cache. CacheCOW contains some elements of TMM, including providing different hit rates to different classes of applications, but targets Internet servers in a theoretic and simulation context. CacheCOW does not address many of the practical issues in using, gathering, and aging LRU histograms.

Resource Containers provide static allocations to resources such as physical memory [5] to prevent interference between different classes of applications. However, such static allocations are inherently ineffective and do not determine what to set the allocations to. We have shown in our evaluation the benefits of using a dynamic scheme, such as TMM, over static allocations. It is possible to adjust these limits at runtime, and therefore would be possible to use the memory tracing and LRU analysis techniques of TMM to adjust resource container limits, or to dynamically adjust other systems which control memory use.

LRU histograms [12], or page recency-reference graphs [11, 9], are useful in many contexts such as memory allocation and virtual memory compression and are essential to TMM. Some optimizations to gathering histograms have been implemented that rely on page protection but lower the overhead to 7-10% [13]. In this paper, we use an adapted form of tracing that avoids the overhead of handling relatively expensive page faults [2].

6 Conclusions

In this paper we present an OS mechanism, the Transparent Memory Manager (TMM), for supporting transparent contribution of memory. This system prevents contribu-

tory applications from interfering with the performance of a user's applications, while maximizing the benefits of harnessing idle resources. TMM protects the user's pages from unwarranted eviction and limits the impact on the performance of user applications to less than 5%, while donating hundreds of megabytes of idle memory.

Notes

¹ Downloadable from: <http://prisms.cs.umass.edu/software.html>

References

- [1] G. Almasi, C. Cascaval, and D. A. Padua. Calculating stack distances efficiently. In *The International Symposium on Memory Management (ISMM 2002)*, Berlin, Germany, June 2002.
- [2] J. Cipar, M. D. Corner, and E. D. Berger. Transparent contribution of storage and memory. Technical Report 06-05, University of Massachusetts-Amherst, Amherst, MA, January 2006.
- [3] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pas-tiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36(SI):285–298, 2002.
- [4] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.
- [5] P. Druschel G. Banga and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, Louisiana, February 1999.
- [6] P. Goyal, D. Jadav, D. Modha, and R. Tewari. Cachecow: Qos for storage system caches. In *International Workshop on Quality of Service (IWQoS)*, Monterey, CA, June 2003.
- [7] Theodore Johnson and Dennis Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 439–450, Santiago, Chile, 1994.
- [8] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th SOSP (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.
- [9] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. The EELRU adaptive replacement algorithm. *Performance Evaluation*, 53(2), July 2003.
- [10] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, November 1994.
- [11] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of The 1999 USENIX Annual Technical Conference*, pages 101–116, Monterey, CA, June 1999.
- [12] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *Proceedings of the Third International Symposium on Memory Management (ISMM)*, Vancouver, October 2004.
- [13] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamically tracking miss-ratio-curve for memory management. In *The Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, October 2004.

Implementation and Evaluation of Moderate Parallelism in the BIND9 DNS Server

JINMEI, Tatuya
Toshiba Corporation
jinmei@isl.rdc.toshiba.co.jp

Paul Vixie
Internet Systems Consortium
vixie@isc.org

Abstract

Suboptimal performance of the ISC BIND9 DNS server with multiple threads is a well known problem. This paper explores practical approaches addressing this long-standing issue. First, intensive profiling identifies major bottlenecks occurring due to overheads for thread synchronization. These bottlenecks are then eliminated by giving separate work areas with a large memory pool to threads, introducing faster operations on reference counters, and implementing efficient reader-writer locks. Whereas some of the solutions developed depend on atomic operations specific to hardware architecture, which are less portable, the resulting implementation still supports the same platforms as before through abstract APIs. The improved implementation scales well with up to four processors whether it is operating as an authoritative-only DNS server, with or without dynamic updates, or as a caching DNS server. It also reduces the memory footprint for large DNS databases. Acceptance of this new server will also have a positive side effect in that BIND9, and its new features such as DNSSEC, should get wider acceptance. The direct result has other ramifications: first, the better performance at the application level reveals a kernel bottleneck in FreeBSD; also, while the results described here are based on our experience with BIND9, the techniques should be applicable to other thread-based applications.

1 Introduction

As the Internet has become an indispensable piece of infrastructure, the domain name system (DNS)[10] has also been facing increasing pressure as a core feature of the Internet. For example, forged DNS data can lead to net scams, and the IETF has standardized a new version of security framework for DNS (DNSSEC[2]) in order to address such threats. At the same time, top level DNS servers have been receiving more queries, and have even

been the target of denial of service attacks. Today's DNS servers thus need to have the latest functionality as well as higher performance to handle the heavy query rate.

ISC BIND9[6] was designed to meet these seemingly contradictory requirements. It supports all standardized DNS-related protocols, including the latest version of DNSSEC as well as adopting a multi-thread architecture so that it could meet the performance requirements by using multiple processors.

Unfortunately, BIND9's multi-thread support did not benefit much from multiple processors. The performance measured by queries per second that the server could process was soon saturated or even degraded as the number of processors and threads increased. In the worse case, BIND9 with multiple threads showed poorer performance compared to a single process of its predecessor, BIND8, which does not even try to take advantage of multiple processors.

Partly due to the poorer performance, operators who see high query rates have tended to stick to BIND8, implicitly hindering wider deployment of new technologies like DNSSEC.

We address the performance problem with multiple threads through a set of practical approaches and have already reported a preliminary result [7] for an authoritative-only DNS server that does not allow dynamic update requests [16]. This paper completes the ongoing work by improving memory management further and providing thorough evaluation and discussions of the implementation. The evaluation covers the case with dynamic updates or caching and with test data based on real DNS traffic.

The rest of the paper is organized as follows. In Section 2, we provide an overview of the BIND9 implementation architecture as a base for later discussions. Section 3 explains how to identify major bottlenecks regarding thread synchronization, and Section 4 describes our approaches to eliminate the bottlenecks followed by some detailed notes about the implementation in Section 5. We

then show evaluation results for the improved implementation in terms of response performance and run-time memory footprint along with relevant discussions in Section 6. We review related work in Section 7 and conclude in Section 8 with remaining work.

2 BIND9 Architecture

In general, DNS servers are categorized as authoritative servers and caching (recursive) servers. An authoritative server has the authority for at least one *zone*, an administrative perimeter within the DNS. When the server receives a DNS query for a domain name, it searches the zone that best matches the queried name, and, if found, responds with the corresponding resource records (RRs) stored in the zone. A caching server handles queries for end stations by forwarding them toward authoritative servers until a determinate answer is received. This process is called *recursive resolution*. The answer is cached for possible re-use, and then forwarded back to the originating end station.

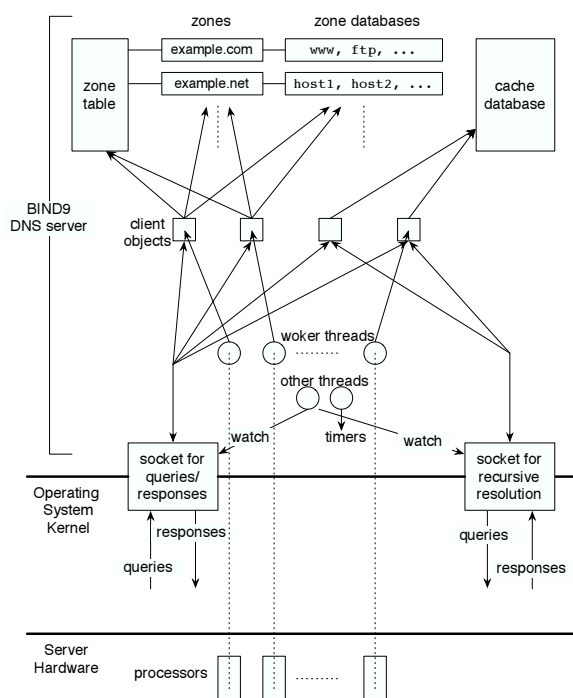


Figure 1: The system architecture of a DNS server with multiple processors running threaded version of BIND9.

Figure 1 is an overview of the entire system of a DNS server that has multiple processors and runs BIND9 with multiple threads.

BIND9 can act as either an authoritative or caching server. For the authoritative service, it has a *zone table*, whose entries correspond to the zones for which the

server has the authority. Each zone entry has an associated *zone database*, which consists of RRs of that zone. For the caching service, it has a separate *cache database* for storing results of recursive resolution. BIND9 can support multiple implementations of databases, but the default implementation of zone databases and the cache database is the same. The zone table, zone databases, and the cache database are, by default, built in-memory, either from configuration files or dynamically.

When built with threads, BIND9 creates a set of *worker threads* at startup time. The number of worker threads is equal to the number of available processors by default, and the threads run concurrently on different processors handling DNS queries. Note that an additional “pool” of worker threads is unnecessary and indeed unused, especially for an authoritative server; since the processing of each query does not involve any network or file I/O, or blocking operations in general, there is no benefit in handling more queries than the number of processors concurrently.

Each DNS query arriving at the server is represented as a separate data structure called a *client object* in the BIND9 implementation. Available worker threads are assigned to the client objects and process the queries by consulting the zone table and the appropriate zone databases. If the server does not have authority for the queried domain name, the worker thread looks for the answer in the cache database. In either case when an answer is found the worker thread sends the response packet to the socket that received the query.

A caching server often needs to send additional queries to external authoritative servers in processing incoming queries. In BIND9 the worker thread sends such queries through the client object and receives the responses for further processing. While waiting for the responses, the worker thread works on other client objects, if any, so that it can handle as many queries as possible.

There are some other threads, regardless of the number of processors, which manage timers or watch events regarding sockets.

Since multiple threads get access to the zone table and databases concurrently, such access must be properly protected to avoid race conditions. BIND9 uses standard POSIX pthread locks [4] for this purpose.

3 Identifying Bottlenecks

The first step in this work was to identify major bottlenecks that affected BIND9’s response performance in multi-processor environments. We used a simple profiler for lock overhead contained in the BIND9 package for this purpose. It is available by building the server with the `ISC_MUTEX_PROFILE` preprocessor variable being

non-0. When enabled, it measures for each lock the waiting period for achieving the lock and the period that the lock is held by issuing the `gettimeofday` system call before and after the corresponding periods. The BIND9 server retains the statistics while it is running, and dumps the entire result on termination as follows:

```
mem.c 720: 1909760 7.223856 24.671104
mem.c 1030: 108229 0.401779 1.625984
mem.c 1241: 67 0.000107 0.000108
...
```

The above output fragment shows the statistics of a lock created at line 720 of `mem.c`. This lock was acquired 108,229 times at line 1030 of `mem.c`, the total time that threads spent in the corresponding critical section was 0.401779 seconds, and the total time that threads waited to acquire the lock was 1.625984 seconds.

We built BIND 9.3.2 on a SuSE Linux 9.2 machine with four AMD Opteron processors and with enabling the lock profiler¹. We then configured it as a root DNS server using a published snapshot of the root zone, sent queries to the server for 30 seconds, and observed its behavior (see Section 6 for more details about the test environment).

According to the profiling results, threads spent about 52 seconds for acquiring locks, which were 43.3% of the total running time of the worker threads (120 seconds with four threads). We examined the results and categorized the points in the code that dominated the waiting periods as follows:

- 54.0% of the total waiting period belonged to memory management routines for allocating or releasing temporary memory to make response packets.
- 24.2% were mainly for incrementing or decrementing reference counters to some data objects.
- 11.4% were for getting access to data objects representing zones or resource records.
- 10.4% took place in BIND9's internal reader-writer lock implementation. These locks were for either the zone table or zone databases.

It may look too severe that nearly a half of the running time was occupied just for acquiring locks. But it is actually not surprising because the processing of DNS queries is lightweight. Especially in the case of an authoritative server, it does not involve any additional network or disk I/O, and the synchronization cost between the threads is relatively much expensive.

4 Eliminating Bottlenecks

In the following subsections, we discuss details about how to eliminate the bottlenecks shown in the previous section.

4.1 Working Space Separation

According to the profiling results, the most significant bottleneck was in memory management routines. Specifically, it was caused by a mutex lock for a data structure called the *memory context*. This structure maintains memory usage information in order to detect a memory leak and contains some small fragments of memory that can be reused. Multiple client objects share a single memory context and use it for building response messages frequently in the original BIND9 implementation, which caused the severe lock contentions.

In addition, the original implementation uses the standard library versions of `malloc` and `free` to allocate and release memory, which may internally cause additional lock overhead.

We mitigated the first type of overhead by separating memory contexts for different client objects. This is safe because the allocated memory is a temporary working space specific to each client and the worker thread working on it, and generally does not have to be shared with other threads.

In order to reduce the overhead imposed by the standard library, we enabled a BIND9's experimental feature called the "internal memory allocator". It initially allocates a 256KB memory block as a large pool for each memory context, and repeatedly uses fragments in the memory pool without involving the standard library.

One obvious issue with this approach is the additional memory consumption for the separate space. Overconsumption of memory is of concern particularly when BIND9 acts as a caching server. In this case, it can have many client objects that are handled by a smaller number of worker threads: by default, the maximum number of concurrent clients is 1000, while the number of worker threads is equal to the number of processors. If we simply separated the work space for each client object, the maximum amount of memory needed might be unacceptably large.

Instead, we had multiple client objects share a single memory context as shown in Figure 2. The number of contexts has a fixed limit regardless of the number of client objects, thereby ensuring the upper limit of required memory for the work space.

Since different worker threads can work on client objects sharing the same memory context, access to the allocated memory must be locked properly. For example, worker threads A and B in Figure 2 share the same mem-

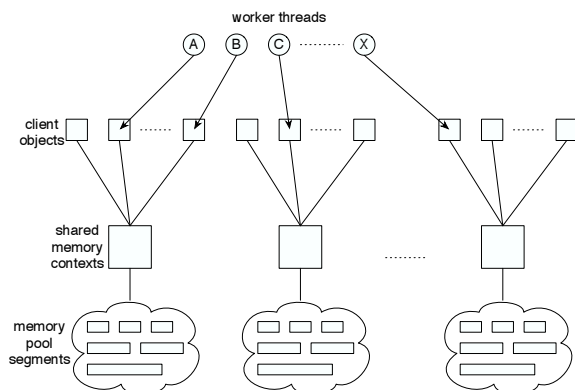


Figure 2: Sharing memory contexts by clients.

ory context and may contend with each other for holding the lock of the context.

We avoided the lock contention in a probabilistic way, that is, by creating many more shared memory contexts on-demand than worker threads. In the current implementation, the maximum number of memory contexts is 100, which we believe to be large enough for a decent number of worker threads in environments where BIND9 is expected to work. Yet this number is an arbitrary choice; in fact, we have not seen significant differences in performance with 10 or 100 clients with evaluation configurations described in Section 6. Its effectiveness will have to be assessed through live experiments.

Regarding the additional memory consumption, the total amount of memory for the preallocated block occupies the dominant part. If all the 100 contexts are created and used, the total amount will be 25MB. More blocks of memory could be required if the clients consume all the pre-allocated memory, but typically it should not happen since the memory is used only for temporary work space to build a moderate size of response message and only requires a small amount of memory. It should also be noted that an authoritative server does not need additional clients, and the discussion only applies to a caching server. For a busy caching server, which generally needs a large amount of memory for the cache, we believe the additional consumption is minor.

4.2 Faster Operations on Counters

As shown in the previous section, nearly a quarter of the major bottlenecks in terms of the waiting period caused by locks were regarding operations on reference counters. Reference counters are essential for threaded programs, since multiple threads can independently refer to a single data object and it is hard to determine without counters whether a shared object can be safely destroyed.

Obviously, change operations on a reference counter must be mutually exclusive. In addition, a significant action for the referenced object can happen when the reference increases from or decreases to zero, in which case additional locks may be necessary to protect that action. However, such an event does not take place in the typical case where worker threads are working on those references, since the base system usually holds the structure in a stable state. For example, a zone keeps a positive reference to the corresponding database until it replaces the database with a new one, e.g., by reloading the configuration file or at shutdown time. Thus, the locks for reference counters generally only protect the simple operation of increasing or decreasing an integer.

This observation led us to the following idea: most of today's processors have dedicated atomic instructions that allow multiple threads to increment or decrement an integer concurrently and atomically. Some processors support an instruction exactly for this purpose. We can also implement these operations with other type of instructions such as compare-and-swap (CAS) and a busy loop, as shown in Chapter 15 of [8]. In either case, the resulting operations on reference counters can run much faster than those protected by general locks, and we will simply call both "atomic operations" hereafter. We still use normal POSIX locks for the rare cases where the counter decrements to zero.

4.3 Efficient Reader-writer Lock

According to the results of Section 3, about 30% of the locks affecting the performance were related to access to the zone table or to zone databases. While some of the locks are general and allow only one thread to enter the critical section at once, the access is read-only in most cases for an authoritative DNS server that does not allow dynamic updates: once the server loads the configuration and zones, the worker threads just search the zone table and zone databases for the answers to queries, without modifying the objects they are referring to.

One possible optimization here is to use reader-writer locks (rwlocks). In fact, the original BIND9 implementation used rwlocks for some cases considered here. It uses a custom version of rwlocks instead of the pthread rwlock library in order to ensure fairness, and the custom implementation relies on the general locks internally. As a side effect of this, readers can be blocked even if there is no writer.

If we could assume some atomic operations supported by the underlying hardware, however, rwlocks could be implemented more efficiently [9]. We adopted a slightly modified version of the "simple" algorithm described in [9]², and used the new rwlocks for all of the above cases. The new rwlock implementation requires an atomic op-

eration that adds some (positive or negative) value to an integer and an atomic CAS operation.

The key to the algorithm are the following integer variables, which are modified atomically: `write_requests`, `write_completions`, and `cnt_and_flag`.

The first two variables act as a waiting queue for writers in order to ensure the FIFO order. Both of them begin with the initial value of zero. When a new writer tries to get a write lock, it increments `write_requests` and gets the previous value of the variable as a "ticket". When `write_completions` reaches the ticket number, the new writer can start writing. When the writer completes its work, it increments `write_completions` so that another new writer can start working. If the `write_requests` is not equal to `write_completions`, it means a writer is now working or waiting. In this case, new readers cannot start reading; in other words, this algorithm prefers writers.

The variable `cnt_and_flag` is a "lock" shared by all readers and writers. This 32-bit integer variable acts as a structure with two members: `writer_flag` (1 bit) and `reader_count` (31 bits). The `writer_flag` bit reflects whether a writer is working, and `reader_count` is the number of readers currently working or almost ready to work (i.e., waiting for a currently working writer).

A writer who has the current "ticket" tries to get the lock by exclusively setting the `writer_flag` to 1, provided that the whole 32-bit value is 0 (meaning no readers or writers working). We need the atomic CAS instruction here. On the other hand, a new reader first checks there are no writers waiting, and then increments the `reader_count` field while getting the previous value of `cnt_and_flag`. If the `writer_flag` bit is not set, then the reader can enter the critical section; otherwise, it waits for the currently working writer.

When the necessary prerequisite conditions are not met, the reader or the writer sleeps until the related conditions change. When a working reader or writer completes its work, some readers or writers are sleeping, and the condition that suspended the reader or writer has changed, then it wakes up the sleeping readers or writers. Our implementation uses condition variables and locks as defined in the standard pthread library for these cases. The use of the standard library may degrade the performance, but we believe this does not matter much, since writers should appear only very rarely in our intended scenarios. In addition, we found the extensions to the original algorithm described below could be implemented much easier with the standard library for the rare cases.

The original algorithm based on [9] was not starvation-free for readers. In order to prevent readers

from starving, our implementation also introduced the "writer quota" (Q). When Q consecutive writers have completed their work, possibly suspending readers, the last writer will wake up the readers even if a new writer is waiting.

We implemented other extensions: "trylock", "tryupgrade", and "downgrade", which are necessary in some special cases in the BIND9 implementation.

The "trylock" extension allows the caller to check whether the specified lock is achieved and to get the lock when possible without blocking. This extension is actually used for writers only. Our implementation of this extension lets the caller pretend to have the "current ticket" and try the same thing as a candidate writer would normally do as described above. If this succeeds, then the caller decrements `write_completions` as if it had the "current ticket", and starts writing. Otherwise, this attempt simply fails without making the caller sleep.

The "tryupgrade" extension allows a reader who already has a lock to become a writer if there are no other readers. The implementation is similar to that of "trylock", but in this case the prerequisite condition for the CAS operation is that `reader_count` be one, not zero, which indicates this is the only reader.

On the other hand, if a writer "downgrades", it becomes a reader without releasing the lock, so that other readers can start reading even if there is a waiting writer. The implementation of downgrading is straightforward and is not discussed here.

4.3.1 Alternative Approach: Standard Rwlock

One possible alternative to the optimized, but less portable rwlock described above is the standard rwlock implementation [15] provided as a part of the operating system. It is clearly advantageous in terms of portability, and it may also provide decent performance depending on the implementation detail.

However, the standard rwlock specification is not guaranteed to be starvation free, which is the primary reason why we did not rely on it. In addition, the performance of the standard implementation varies among different implementations as we will see in Section 6.1.6. Considering the variety of architectures that have the necessary atomic operations for the customized rwlock implementation (see Section 5.2), we believed our approach with the customized implementation would provide better performance for as many systems as possible.

5 Implementation

We modified BIND9 using the optimization techniques described in the previous section and contributed the new implementation to ISC. While it is not publicly available

as of this writing, it has been incorporated in the ISC's development tree, and will be released as free software in BIND 9.4, the next major release of BIND9.

In the following subsections we make some supplemental notes on the implementation.

5.1 Where and How to Use Rwlocks

It was not straightforward to apply the efficient rwlock implementation described in Section 4.3. We used the rwlocks for the zone table, each of the zone and cache databases, and every database entry, which is sets of RRs (called *RRsets*) for the same name.

The original implementation used normal mutex locks for the RRsets, not rwlocks. In order to use the rwlocks for the RRsets in an effective way, we first separated the targets protected by the locks into reference counters and other content. We then ensured safe and efficient operations on the former as described in Section 4.2 while using rwlocks for protecting access to the latter. This way we typically only need read-only access to the RRsets, where the rwlocks work effectively. This is definitely the case for authoritative-only servers that do not allow dynamic updates.

For caching servers, which need write access to RRsets in the cache database more frequently, we introduced one further customization. Whether it is a zone or cache database, the original implementation used a lock bucket, each entry of which contained a lock shared by multiple RRsets in order to limit the necessary memory for the locks. Moreover, the number of entries in a bucket was set to 7 by default, a pretty small number, so that the memory consumption would still be small even if the server had a large number of zones. Using a lock bucket should make sense, but it was not reasonable for applying the same limit of the bucket size to zone databases and the cache database because there is typically only one cache database in a single server. Thus, we enlarged the bucket size for the cache database to a much larger number. It is 1009 in the current implementation, which is an experimental arbitrary choice at the moment.

5.2 Portability Considerations

Some approaches described so far rely on hardware dependent atomic operations. An obvious drawback of such approaches is that the resulting implementation becomes less portable. This is particularly severe for BIND9, since portability is one of its major goals as a reference implementation.

To ensure as much portability as possible, we minimized the necessary operations. In fact, we only need two operations: atomic addition on an integer, which can also be used for atomic operations on counters, and

atomic CAS. Furthermore, these can actually be emulated using other primitive operations.

We have implemented the two operations, through emulation in some cases, for Intel x86, AMD, Alpha, Sparc v9, PowerPC, and MIPS as a proof of concept. The first two have dedicated instructions for our purposes [5]. Sparc v9 has CAS as a primitive instruction [13], on which we can emulate the atomic addition on an integer. The others have a variant of locked load and store instructions, with which we can emulate the necessary operations. We believe these operations can also be implemented on many of today's architectures.

However, we still expect the operations cannot be provided in a realistic way for some architectures. Thus, any use of these operations are hidden under an abstract interface, and the new code works just as before on machines that do not support the necessary operations for our optimizations.

5.3 Optimizing OS Kernel

Application-level optimization sometimes reveals system bottlenecks. In fact, after implementing the possible optimizations described so far, we noticed that FreeBSD still did not show the performance we expected on a fast multi-processor machine, while Linux and Solaris showed the anticipated improvement.

We then examined the kernel source code of FreeBSD and found that a kernel lock was protecting the send socket buffer. The lock was to protect simultaneous access to parameters of the socket buffer such as watermarks for flow control or a list of outgoing packets kept in the socket for retransmission. This lock was held throughout the socket output routine including the output processing at the network and datalink layers. Since a DNS server typically works on a single UDP port and the worker threads of BIND9 share a single UDP socket for sending responses as a result, they would contend in the kernel attempting to acquire the lock.

For a UDP socket, however, this lock was unnecessary. In fact, none of the protected parameters were used for the output processing of UDP packets due to its property as a non-reliable transport protocol without flow control.

We wrote a simple patch that omitted the unnecessary lock for UDP sockets, and confirmed that it worked efficiently without causing any disruption in the kernel. We then reported this performance issue with the patch to the FreeBSD developers community. Fortunately, FreeBSD's netperf project [17] has removed this bottleneck in a recent change based on our report.

6 Evaluation

We evaluated the effectiveness of the implementation described so far using a real multi-processor machine and actual DNS queries.

The base of our implementation was an unpublished version of BIND9. The response performance we achieved, however, should be equivalent to BIND 9.3.2, the latest released version as of this writing.

The target machine used for the evaluation was a 4-way AMD Opteron with 3.5GB of memory, 1GB of L2 cache, and 64KB of L1 cache (64KB each for instruction and data) and had a Broadcom BCM5704C Dual Gigabit Ethernet adaptor.

Most of the results shown in this paper were those for SuSE Linux 9.2 (kernel 2.6.8 and glibc 2.3.3) running in the 64-bit mode, unless explicitly noted otherwise. We also performed the same tests against FreeBSD 5.4-RELEASE and Solaris 10 on the same hardware, both running in 32-bit mode, and confirmed the overall results were generally the same.

We built BIND9 both with and without the optimizations we developed on the test machine, which we call “BIND9 (old)” and “BIND9 (new)”, respectively. For comparison, we also used BIND9 without enabling threads and BIND 8.3.7. These are referenced as “BIND9 (nothread)” and “BIND8”.

6.1 Response Performance

6.1.1 Server Configurations

We configured various types of DNS servers for evaluating the response performance of our implementation. These configurations have several different characteristics.

The first type of configuration is for “static” authoritative servers, i.e., authoritative-only DNS servers that do not allow dynamic updates, which has the following, three specific configurations:

- A root server configuration with a real snapshot of the root zone as of October 28th, 2005. Snapshots of the root zone are publicly available at <ftp://ftp.internic.net/>. We specifically emulated the F-root server, which also had authority for the “.ARPA” zone. We used a copy of the “.ARPA” zone data of the same day from the F-root server via the zone transfer protocol and used it for configuring the test server.
- The “.NET” server configuration with a copy of the actual zone database as of March 2003. It contained 8,541,503 RRs and was used as a sample of a single huge zone. The vast majority of the zone data consists of NS and glue RRs.

- A generic server configuration with a massive number of small zones. Specifically, it had 10,000 zones, each of which contained 100 A RRs in addition to a few mandatory RRs for managing the zone.

The second type of configuration is for a “dynamic” authoritative server. It started with a single pre-configured zone containing 10,000 RRs in addition to a few mandatory RRs and accepted dynamic update requests.

The third type of configuration is for a “caching” recursive server. It started without any authoritative zones and accepted recursive queries.

6.1.2 Evaluation Environment

Our measure of performance common to all the configurations was the maximum queries per second that the tested implementation could handle without dropping any queries. Our engineering goal regarding this measure was to add 50% of the single-processor query rate for every additional processor. This engineering goal was set so that BIND9 with two processors would outperform BIND8 if BIND9 with a single processor could operate at no less than 80% of the speed of BIND8 on that same single processor. While the target may sound conservative, we believe this is in fact reasonable since the base performance of BIND9 is generally poorer than BIND8 due to its richer functionality, such as support for DNSSEC, and the cost of securer behavior, such as normalization of incoming data.

We connected two external machines to the same Gigabit Ethernet link to which the test machine was attached. The external machines acted as the clients for the evaluation target. In order to avoid the situation where performance on the client side was the bottleneck, we used multiple processes on the client machines.

We used a variant of the `queryperf` program contained in the BIND9 distributions for the client side processes, which was slightly customized based on the original tool for our testing purposes. The program was modified so that it could combine the results of multiple processes on different machines and could send dynamic update requests as well as ordinary queries. In all test cases `queryperf` repeatedly sent pre-configured queries, adjusting the query rate in order to avoid packet loss, and dumped the average queries per second processed.

6.1.3 Static Server Performance

We prepared the test queries for the root server configuration based on real traffic to the F-root server located in San Francisco, California. We used packets that had reached the server between 5:18am and 6:18am on Oc-

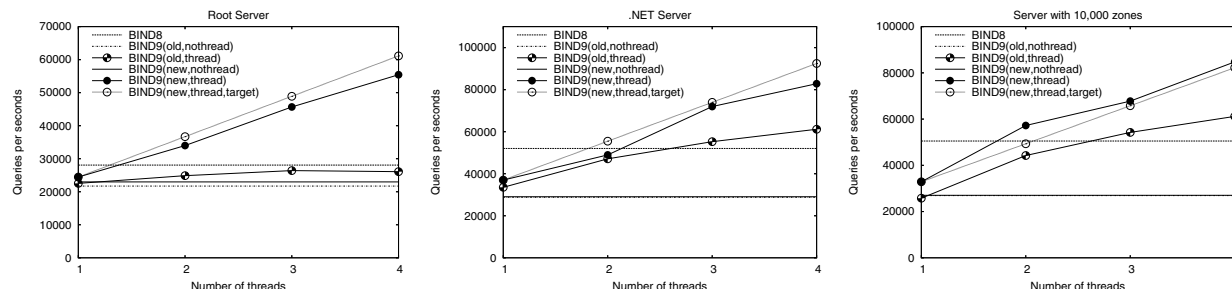


Figure 3: Evaluation Results for the static configurations.

tober 28th, 2005, a sample which contained 31,244,014 queries.

The reason why we performed the evaluation based on real data was because details of the query pattern may affect the response performance significantly. Of particular importance in terms of bottlenecks for a multi-threaded BIND9 server is the proportion of queries that match the same database entries; if a large fraction of the whole queries matches a small number of entries, it may cause contentious access by multiple threads and can be a bottleneck due to the contentions for acquiring locks. In fact, we analyzed the query data and found that the query pattern was unbalanced: 22.9% of the queries was for names under the same domain (.BR) and names under the top six domains occupied more than a half of the whole queries.

We separated the queries into three chunks and ran three instances of the `queryperf` program concurrently with the divided queries as input so that the resulting test query stream roughly emulated the actual query traffic. We did not stick to reproducing the real queries in the same order and at the same timing since our primary goal was to know the maximum performance of the tested implementation.

For all other authoritative server configurations than the root server case, we generated the test queries by randomly choosing domain names under the zone that the target server managed. A small fraction of the query names did not exist in the zone and resulted in negative responses.

Figure 3 summarizes the evaluation results. For BIND8, and for BIND9 with “nothread”, the results for more than one thread are meaningless. For comparison purposes, however, we showed these meaningless results as if they were independent of the number of threads. The graphs labeled “target” show our original engineering goal.

Although we did not reach our original engineering goals in some cases, our results are generally good. The new BIND9 implementation also scaled almost linearly to the number of processors, up to a maximum of four

processors.

In comparison to BIND8, the new BIND9 implementation with two processors could answer the same number or more queries than BIND8 could, and with three or more processors the results were much better than BIND8. We also note that the old BIND9 implementation could never outperform BIND8 even with all the four processors for the root server configuration.

Another remarkable point is that BIND9 with one thread could generally handle more queries than BIND9 without threads. This is not an intuitive result, since both implementations should benefit only from a single processor and there should be overhead specifically related to threads. We discovered the reason for the result was because the “nothread” version needed to check other asynchronous events such as timer expiration periodically, even though such events rarely occurred; in the threaded version, the worker thread could keep handling queries without any interrupts, since separate threads which were effectively idle could handle the exceptional events. This should prove that a well-tuned threaded application can run more efficiently even on a single processor machine.

6.1.4 Performance with Dynamic Updates

To evaluate the authoritative server while it was allowing dynamic updates, a separate client process sent update requests at a configured rate, cooperating with the `queryperf` processes. In order to emulate a busy authoritative server handling frequent update requests, we measured the total response performance using the update rates of 10 and 100 updates per second.

Figure 4 is the evaluation results of these scenarios. There is an exception in the case of BIND9 implementations receiving the higher rate of update requests with no or one thread: the BIND9 server using a single processor could only handle at most 75 updates per second. With two or more processors, it could process the configured update rate.

In either case the new BIND9 implementation

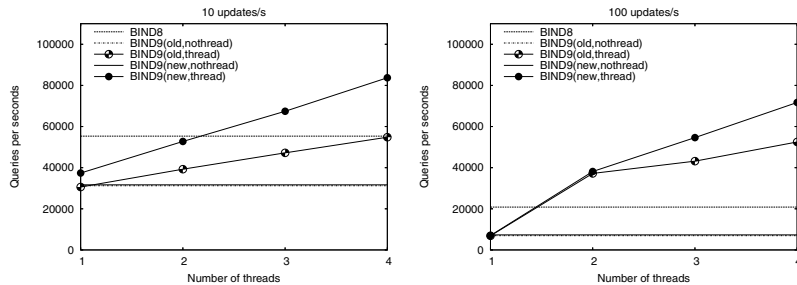


Figure 4: Evaluation results for a server accepting 10 and 100 update requests per second.

achieved the expected improvement in that it could at least handle with two threads as many queries as BIND8 and could process more with three or more threads. This result proves that the efficient rwlock implementation works as expected with a small number of writers.

It should be noted that dynamic update processing cannot benefit from multiple processors, since update requests must be handled sequentially. We thus fixed the update rate regardless of the number of threads in the evaluation, which means the total queries handled with one thread is much smaller than the other cases. Therefore, the “target” performance in these cases do not matter much, and we did not include it in Figure 4.

6.1.5 Caching Server Performance

It is not trivial to evaluate the response performance of a caching DNS server as pointed out by [12]. The main reason is that query processing at a caching server can involve additional transactions with external authoritative servers and so the total performance depends on various factors such as available network bandwidth, the response performance or availability of the external authoritative servers, or the cache hit rates. Some of the dominant factors are hard to control or reproduce.

We used a simplified model for our evaluation as shown in Figure 5. It consisted of two external authoritative servers in addition to a client (tester) and the caching server (evaluation target), all attached to the same Ethernet segment. External server 1 had authority for 1,000 external zones, each of which contained 200 RRs, and external server 2 had authority for a common parent zone of these zones and delegated the authority of the child zones to server 1. The client sent queries for names belonging to these zones to the caching server.

In the initial stage of the performance evaluation the caching server needed to follow the delegation chain of authority from the root server to external server 1. The caching server then repeatedly queried external server 1 and cached the result as it received queries from the client (as shown by exchanges 1 through 4 in the figure). At

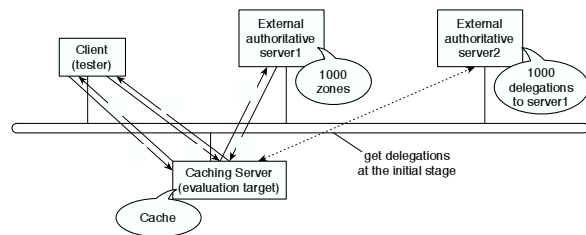


Figure 5: Network configuration for evaluating caching server performance.

some point the cache stabilized, and the caching server did not have to query the external server as long as the data remained in the cache (as shown by exchanges 5 and 6 in the figure). At this stage we measured the response performance.

The configurable evaluation parameter in this test scenario was the cache hit rate. We configured half of the RRs stored in external server 1 with a Time-to-live (TTL) of 1 second so that it would be highly unlikely to be reused in the cache. If the caching server received a query whose answer had expired during the test period, the caching server queried the external server again. We then prepared various test query patterns for specific hit rates by mixing the names with the shorter TTL and the other names in an appropriate ratio.

Our primary target of cache hit rate was 80% hits. This number was based on a statistic analysis of an existing busy caching server that had a large number of clients. We also used other query data that caused the hit rates from 50% to 90% for comparison.

Figure 6 shows the evaluation result for the caching server configuration with the primary target of cache hit rate. While the new BIND9 implementation scaled well compared to the target performance, and it could answer more queries with four threads than BIND8, this was not fully satisfactory in that it needed all four threads (processors) to outperform BIND8.

It should be noted, however, that BIND9 with one thread could only handle 43% of queries that BIND8

could answer. This means that the fundamental reason for the poorer performance is the base processing ability with a single thread, as explained in Section 6.1.2, rather than bottlenecks in the multi-thread support. It should be more reasonable to consider improving the base performance first in order to achieve competitive performance with multiple threads in this case.

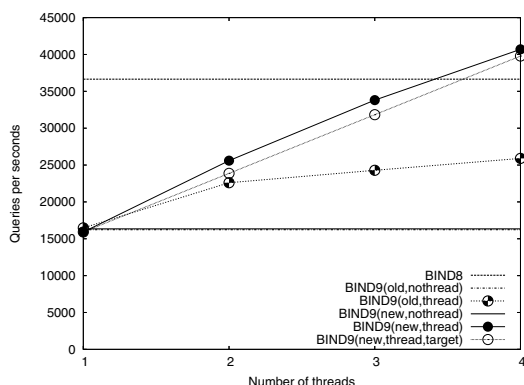


Figure 6: Evaluation Results for a caching server with 80% cache hits.

Table 1 summarizes the results of the same evaluation with various cache hit rates. It generally indicates the same result as shown in Figure 6 regardless of the hit rates: the new BIND9 implementation scaled well with multiple threads, but the resulting performance compared to BIND8 was not enough due to poorer base performance.

Rate (%)	BIND8	BIND9 (new) with threads			
		1	2	3	4
90	41393	20569 (50%)	31287 (76%)	42826 (103%)	51686 (125%)
80	36647	15915 (43%)	25595 (70%)	33809 (92%)	40697 (111%)
70	32660	13388 (41%)	21638 (66%)	27561 (84%)	33269 (102%)
60	29862	10296 (35%)	18059 (60%)	22834 (76%)	28100 (94%)
50	29676	9641 (32%)	15610 (53%)	19713 (66%)	24391 (82%)

Table 1: A summary of performance evaluation on caching server implementations with various cache hit rates (the left-most column). The numbers for the BIND8 and BIND9 columns indicate the maximum number of queries that the corresponding implementation could handle per second. The percentage numbers for the BIND9 columns are the ratio to the corresponding result of BIND8. The results of BIND9 better than BIND8 are highlighted in a bold font.

6.1.6 Comparison with Alternatives

We also performed additional tests with some of the above configurations for comparing several implementation options. The primary goal in these tests was to see whether the standard rwlocks can be used as a practical alternative as discussed in Section 4.3.1.

Figure 7 shows a summary of the performance comparison. All the optimization types utilize the portable technique of separate memory contexts described in Section 4.1. Note that optimization types (3) and (4) do not rely on hardware dependent atomic operations and are more portable than others. It should also be noted that database entries are protected by mutex locks, not rwlocks, in this type of optimization because the use of rwlocks for the database entries needs faster operations on reference counters as we explained in Section 5.1.

Figure 7 provides several lessons regarding the trade-offs between the optimization techniques.

First, the difference between cases (A) and (B) shows the efficiency of the standard rwlocks varies among different implementations when only reader locks are necessary in the typical case. This means a portable application running on various systems that requires higher performance cannot always rely on the standard rwlock implementation.

Secondly, the result of case (A) indicates that simply using efficient rwlocks may not be enough in terms of total performance. In fact, the difference between optimization types (1) and (2) shows the standard rwlock library is as efficient as our customized version. The comparison between these two and optimization type (3), however, proves that we still need the help of atomic operations to achieve the required performance. One possible explanation of the difference is that type (3) had to use normal mutex locks shared in a small bucket for protecting access to database entries, causing heavy contentions. However, the comparison between types (3) and (4) rather proves that this is likely due to the unbalanced query pattern to the server as explained in Section 6.1.3.

Since the major benefit of implementation (3) is better portability by avoiding machine dependent operations while realizing better performance, this result indicates that attempting to use standard rwlocks cannot completely achieve that goal.

Finally, test cases (C) and (D) seem to show it is not very advantageous to use efficient rwlocks, whether they are based on atomic operations or the standard implementation. In fact, even optimization type (4) worked nearly as efficient as type (1). This is likely because efficient rwlocks were less effective in this case due to the more frequent write operations and because access to cache database entries was less contentious thanks to the

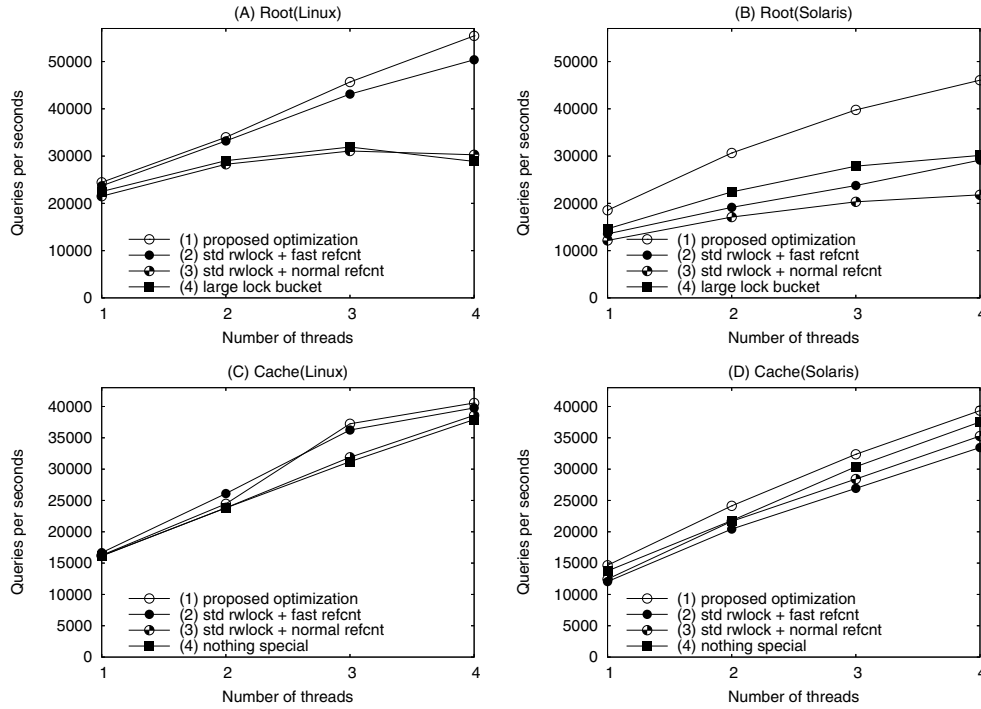


Figure 7: Performance comparison between various types of optimization for the root and caching servers on Linux and Solaris. Optimization types (1) to (3) are common to all the test cases: (1) is the proposed version in this paper; (2) utilizes the standard rwlocks (Section 4.3.1) and faster operations on reference counters; (3) uses the standard rwlocks and normal reference counters. For the root server configuration, optimization type (4) uses a larger lock bucket for zone databases but does not rely on other optimizations, while type (4) for the caching server configuration does not benefit from any optimization.

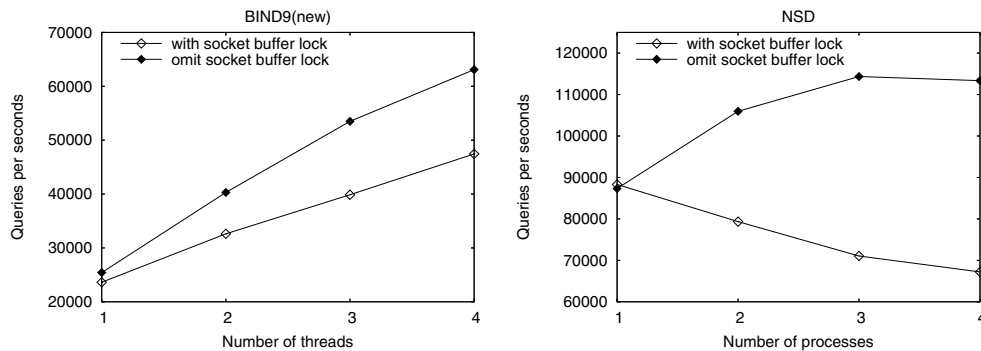


Figure 8: Evaluation of the FreeBSD kernel optimization for the root server configuration.

larger lock bucket.

It should be noted, however, that the actual query pattern may change the result. We generated the query input by randomly and equally choosing names from a large set in these tests, which means query names were well-distributed and less likely to cause contention when acquiring locks. If the real query pattern is more unbalanced as we saw in the root server case, it may cause severe contention that can make a more visible difference in the response performance depending on the rwlock implementation. In fact, we believe this to be the case even for a caching server. For example, the host names of popular search engines stored in the cache database may be queried much more frequently than others. We need further tests with input data which better emulates the real traffic in order to get a definitive conclusion.

6.1.7 Effect of Kernel Optimization

Finally, we verified whether the kernel optimization for FreeBSD described in Section 5.3 made a notable difference with actual query traffic. In addition to the optimized version of BIND9, we used NLnet Labs' NSD[11] for this test. Unlike BIND9, NSD forks multiple processes for concurrent operation, all of which share a single UDP socket. By using NSD, we were able to eliminate the possibility of thread-related bottlenecks altogether.

Figure 8 shows the evaluation results with the root server configuration with or without the lock for the UDP socket output buffer. We omitted the results of the other authoritative server scenarios, but they generally indicated the same conclusion. We also did not show the caching server case because the possible performance was not so high to reveal the kernel bottleneck.

The evaluation results clearly show that the unnecessary lock was a severe bottleneck for a high-performance application that shares a single UDP socket among multiple threads or processes. In particular, the result for NSD indicates the bottleneck can even degrade the performance with a larger number of processes.

While the performance of NSD was saturated with three or more processors, it was not specific to FreeBSD; we saw similar results on Linux and Solaris. We did not figure out the reason for this, but speculated this was due to some lower-level bottlenecks such as in the physical network interface or in memory access contentions.

We did not directly compare the performance between BIND9 and NSD in this test. In fact, the comparison would not make much sense since NSD concentrates on achieving higher response performance by omitting some richer functionality that BIND9 can provide. We will give a qualitative comparison between these two implementations in Section 7.

6.2 Memory Footprint

There is no protocol dictated upper limit of the size of a DNS zone. In fact, some top level zones contain more than 10 million RRs. As a result, the size of DNS zone databases for a server implementation can be huge. For implementations that store the databases in-memory, including BIND, run-time memory footprint is thus critical.

Our implementation adds a new structure field to a set of RRs for faster operations on reference counters, and could increase the memory footprint proportional to the number of the sets, which is typically proportional to the number of RRs. Additionally, we use the efficient version of rwlocks for more data objects than the original implementation did. Since a rwlock generally requires more memory than the normal lock, this could also be a source of larger memory footprint.

We therefore assessed the needed memory for some typical cases which require more memory: a case of a huge zone and a case of 10,000 zones, which were actually the second and third “static” configurations described in Section 6.1.1.

Table 2 summarizes the results. Against our expectation, the new implementation showed even better results than the old one in terms of memory footprint. We thus enabled BIND9's internal memory allocator (see Section 4.1) for the old implementation as well, and compared the results, which are also included in Table 2. It likely indicates the internal allocator recycles memory fragments effectively and reduces the total memory consumption. This also means that the use of internal allocator in our approach makes up for the additional memory consumption used in reducing the lock overhead.

Config	FreeBSD(32bit)		Linux(64bit)	
	Old	New	Old	New
“.net”	762(562)	583	907(802)	811
10K zones	174(143)	164	230(200)	221

Table 2: Run-time memory footprint of authoritative servers of the old and new BIND9 implementations (in MB). Numbers surrounded by parentheses are the footprints for the old implementation enabling internal memory allocator.

7 Related Work

A major subject of our work is to make synchronization among multiple threads more efficient. This is a well-understood research topic, and, indeed, the efficient reader-writer lock implementation we described in Section 4.3 was based on a simple and “naive” algorithm mentioned in old work[9].

As summarized in [1], the current trend of this research area is to pursue scalability with a much larger number of processors, especially using the notion of local spinning. From a practical point of view, however, the scalability aimed for in the theoretical field is far more than enough. In fact, even 4-way machines are not common in such practical areas as DNS operation. We showed in the previous section that our approach scaled well up to the reasonable upper limit.

Meanwhile, such scalable solutions tend to need more complex data structures, requiring more memory, and in some cases more atomic operations. Since some of the data objects we want to protect are sensitive to memory consumption, complex data structures are not really appropriate even if those provide highly scalable performance. Also, requiring more primitive operations damages portability, which is not acceptable from a practical point of view.

Whereas we adopted the built-in memory allocator of the BIND9 implementation for managing temporary work space as explained in Section 4.1, there are other scalable allocators intended for multithreaded applications such as Hoard[3]. It utilizes per-thread heaps to which a set of memory blocks are assigned as a dedicated memory pool for the corresponding threads. The use of per-thread heaps helps avoid “false sharing” (where multiple threads share data on the same CPU cache), yet it limits memory consumption regardless of the number of processors by recycling unused blocks depending on the total usage. In theory, Hoard is better than the BIND9 allocator since the latter can cause false sharing when multiple threads share a single memory context.

However, when we repeated the evaluation tests with BIND9, linking Hoard as well as enabling other optimizations³, we found that it actually performed slightly worse than BIND9’s internal allocator: the former ran 13.1% slower than the latter in the scenario of a caching server with 50% cache hits, which should be the severest test case for a memory allocator. This was probably because the scalability advantage of Hoard did not outweigh its internal complexity; we will have to evaluate the performance with a larger number of processors for fair comparison.

Regarding server implementations, the Apache HTTP server[14] uses atomic operations for incrementing and decrementing reference counters. To the best of our knowledge, however, specific performance evaluation has not been publicly provided. It is also not clear whether the introduction of the atomic operations was based on performance analysis. On the other hand, we first identified operations on reference counters were actually a severe bottleneck through profiling, and showed it could be resolved by introducing an atomic operation.

NSD[11] is another example of high performance

DNS server. It makes the query processing fast by pre-computing the image of response packets for typical queries at initialization time, assuming “static” zone configurations. NSD can also benefit from multiple processors by forking multiple processes for concurrent operation. Indeed, it scales well with multiple processors as we saw in Section 6.1.7. However, it has its own drawbacks. Since each process has an independent copy of data in memory, NSD cannot allow a part of a zone to be dynamically modified via the DNS dynamic update protocol or act as a caching server. Additionally, this approach is not suitable for managing a huge zone because the total memory footprint needed is proportional to the number of processors. Overall, the difference between BIND9 and NSD is a design tradeoff between richer functionality such as the support for dynamic update and higher possible performance in some limited environments.

8 Conclusions and Future Work

We explored several practical approaches for improving the responsiveness of the ISC BIND9 DNS server with multiple threads. We first identified major bottlenecks occurring due to overhead related to thread synchronization through intensive profiling. We then eliminated all the bottlenecks by giving separate work areas to threads using the notion of shared memory contexts, introducing faster operations on reference counters, and implementing efficient reader-writer locks (rwlocks). While some of the solutions developed depended on atomic operations specific to hardware architecture, which are less portable, the resulting implementation still supported the same platforms as before through abstract APIs. We confirmed our implementation scaled well with up to four AMD processors under various configurations from authoritative-only to caching, and with or without allowing dynamic update requests.

Our primary contribution is performance improvements in BIND9, a long-standing issue with that version which has prevented wider deployment. We also hope, as a consequence of these results, that this will promote deployment of new protocol features which previous major versions of BIND do not support, such as DNSSEC.

While the approaches we adopted specifically targeted one particular implementation, we believe our approach includes several lessons that can also help develop other thread-based applications. First, even the seemingly naive approach for identifying bottlenecks in fact revealed major issues to be fixed as shown in Section 3. Secondly, the fact that operations on reference counters were a major bottleneck is probably not specific to BIND9, since these are inherently necessary for objects shared by multiple threads. Thus, our approach to im-

prove the performance for this simple and particular case will help other applications. Finally, the efficient implementation of rwlocks and the framework of shared memory pools can easily be provided as a separate library, not just a subroutine specific to BIND9, and help improve performance of other applications such as HTTP servers or other general database systems than DNS.

We also identified a bottleneck in the UDP output processing of the FreeBSD kernel through our attempt of improving and evaluating the target application and provided a possible fix to the problem. Other applications that benefit from the techniques we provided in this paper may also be able to reveal other bottlenecks hidden inside the system so far.

Even though we proved the effectiveness of our approach through pragmatic evaluation, there may be issues in the implementation which can only be identified with further experiments in the field. In particular, we need feedback on other machine architectures than that we used in Section 6, especially about scalability with a large number of processors, to assess the cost of emulation mentioned in Section 5.2. A larger number of processors may also reveal performance issues in the memory allocator, and will give a more reasonable comparison with a scalable allocator such as Hoard.

Some tuning parameters we introduced are currently an arbitrary choice (Sections 4.1 and 5.1), and we will need to evaluate their effectiveness. We also need more realistic tests for the caching server configuration in order to determine the most reasonable optimization technique regarding both performance and portability, as we discussed in Section 6.1.6. We will continue working on those areas, identifying and solving issues specific to such cases.

Acknowledgments

This work was supported by the Strategic Information and Communications R&D Promotion Programme (SCOPE) of the Ministry of Internal Affairs and Communications, Japan. It was also partially supported by NSF research grant #SCI-0427144.

We thank our shepherd, Vivek Pai, and the anonymous reviewers for their valuable comments and suggestions. We are also indebted to ISC engineers, especially to Mark Andrews, Rob Austein, João Damas, Michael Graff, David Hankins, Peter Losher, and Evan Zalyss-Geller for their support of this work.

Finally, we thank external contributors. Chika Yoshimura helped us in evaluating the implementation. Robert Watson told us details about FreeBSD's SMP support and also incorporated our proposed fix to eliminate an unnecessary kernel lock. George Neville-Neil

and Rick Jones carefully read a draft of this paper and provided valuable comments.

References

- [1] ANDERSON, J. H., KIM, Y.-J., AND HERMAN, T. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing* 16, 2-3 (September 2003), 75–110.
- [2] ARENDS, R., AUSTEIN, R., LARSON, M., MASSEY, D., AND ROSE, S. DNS Security Introduction and Requirements. RFC 4033, March 2005.
- [3] BERGER, E. D., MCKINLEY, K. S., BLUMOF, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multi-threaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)* (November 2000), pp. 117–128.
- [4] IEEE. Information Technology – Portable Operating System Interface (POSIX) – 1003.1c pthreads, 1995.
- [5] INTEL CORPORATION. The IA-32 Intel(R) Architecture Software Developer's Manual. Available from <http://www.intel.com/>.
- [6] INTERNET SYSTEMS CONSORTIUM. ISC home page. <http://www.isc.org/>.
- [7] JINMEI, T., AND VIXIE, P. Practical Approaches for High Performance DNS Server Implementation with Multiple Threads. The Seventh Workshop on Internet Technology (WIT2005).
- [8] KLEIMAN, S., SHAH, D., AND SMAALDERS, B. *Programming With Threads*. Prentice Hall, 1996.
- [9] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming* (April 1991), pp. 106–113.
- [10] MOCKAPETRIS, P. DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION. RFC 1035, November 1987.
- [11] NLNET LABS. NSD home page. <http://www.nlnetlabs.nl/nsd/>.
- [12] NOMINUM, INC. How to Measure the Performance of a Caching DNS Server. Available from <http://www.nominum.com/>.
- [13] SUN MICROSYSTEMS. SPARC Assembly Language Reference Manual. Available from <http://docs.sun.com/>.
- [14] THE APACHE SOFTWARE FOUNDATION. Apache HTTP server home page. <http://httpd.apache.org/>.
- [15] THE OPEN GROUP. The Single UNIX Specification, Version 2, System Interface & Headers (XSH), Issue 5, 1997.
- [16] VIXIE, P., THOMSON, S., REKHTER, Y., AND BOUND, J. Dynamic Updates in the Domain Name System (DNS UPDATE). RFC 2136, April 1997.
- [17] WATSON, R. Introduction to Multithreading and Multiprocessing in the FreeBSD SMPng Network Stack. In *EuroBSDCon 2005*.

Notes

¹We originally began this work with a different OS and machine architecture, but we showed newer results for consistency with the evaluation environment described in later sections.

²The version we used is described in the web page of authors of [9].

³We did this test with Solaris 10, which was the only platform we successfully linked the Hoard library to BIND9.

Flux: A Language for Programming High-Performance Servers

Brendan Burns Kevin Grimaldi Alexander Kostadinov Emery D. Berger Mark D. Corner

*Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003*

{bburns, kgrimald, akostadi, emery, mcorner}@cs.umass.edu

Abstract

Programming high-performance server applications is challenging: it is both complicated and error-prone to write the concurrent code required to deliver high performance and scalability. Server performance bottlenecks are difficult to identify and correct. Finally, it is difficult to predict server performance prior to deployment.

This paper presents Flux, a language that dramatically simplifies the construction of scalable high-performance server applications. Flux lets programmers compose off-the-shelf, sequential C or C++ functions into concurrent servers. Flux programs are type-checked and guaranteed to be deadlock-free. We have built a number of servers in Flux, including a web server with PHP support, an image-rendering server, a BitTorrent peer, and a game server. These Flux servers match or exceed the performance of their counterparts written entirely in C. By tracking hot paths through a running server, Flux simplifies the identification of performance bottlenecks. The Flux compiler also automatically generates discrete event simulators that accurately predict actual server performance under load and with different hardware resources.

1 Introduction

Server applications need to provide high performance while handling large numbers of simultaneous requests. However, programming servers remains a daunting task. Concurrency is required for high performance but introduces errors like race conditions and deadlock that are difficult to debug. The mingling of server logic with low-level systems programming complicates development and makes it difficult to understand and debug server applications. Consequently, the resulting implementations are often either lacking in performance, buggy or both. At the same time, the interleaving of multiple threads of server logic makes it difficult to identify performance bottlenecks or predict server performance prior to deployment.

This paper introduces *Flux*, a domain-specific language that addresses these problems in a declarative, flow-oriented language (Flux stems from the Latin word

for “flow”). A Flux program describes two things: (1) the flow of data from client requests through nodes, typically off-the-shelf C or C++ functions, and (2) mutual exclusion requirements for these nodes, expressed as high-level *atomicity constraints*. Flux requires no other typical programming language constructs like variables or loops – a Flux program executes inside an implicit infinite loop. The Flux compiler combines the C/C++ components into a high performance server using just the flow connectivity and atomicity constraints.

Flux captures a programming pattern common to server applications: concurrent executions, each based on a client request from the network and a subsequent response. This focus enables numerous advantages over conventional server programming:

- **Ease of use.** Flux is a declarative, implicitly-parallel coordination language that eliminates the error-prone management of concurrency via threads or locks. A typical Flux server requires just tens of lines of code to combine off-the-shelf components written in sequential C or C++ into a server application.
- **Reuse.** By design, Flux directly supports the incorporation of unmodified existing code. There is no “Flux API” that a component must adhere to; as long as components follow the standard UNIX conventions, they can be incorporated unchanged. For example, we were able to add PHP support to our web server just by implementing a required PHP interface layer.
- **Runtime independence.** Because Flux is not tied to any particular runtime model, it is possible to deploy Flux programs on a wide variety of runtime systems. Section 3 describes three runtimes we have implemented: thread-based, thread pool, and event-driven.
- **Correctness.** Flux programs are type-checked to ensure their compositions make sense. The atomicity constraints eliminate deadlock by enforcing a canonical ordering for lock acquisitions.
- **Performance prediction.** The Flux compiler optionally outputs a discrete event simulator. As we

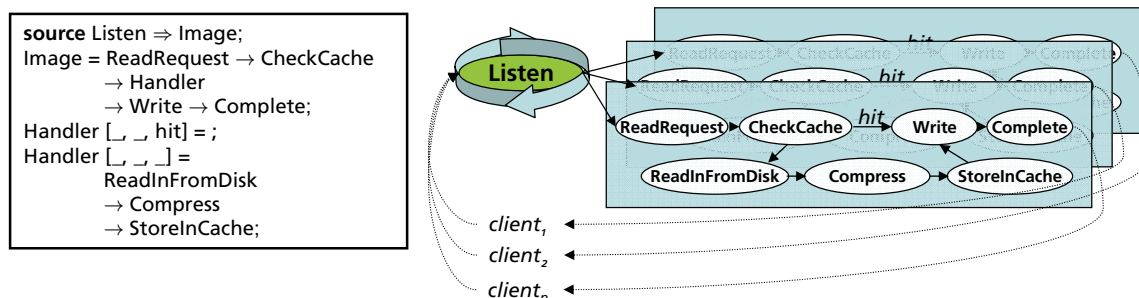


Figure 1: An example Flux program and a dynamic view of its execution.

show in Section 5, this simulator accurately predicts actual server performance.

- **Bottleneck analysis.** Flux servers include lightweight instrumentation that identifies the most-frequently executed or most expensive paths in a running Flux application.

Our experience with Flux has been positive. We have implemented a wide range of server applications in Flux: a web server with PHP support, a BitTorrent peer, an image server, and a multi-player online game server. The longest of these consists of fewer than 100 lines of code, with the majority of the code devoted to type signatures. In every case, the performance of these Flux servers matches or exceeds that of their hand-written counterparts.

The remainder of this paper is organized as follows. Section 2 presents the semantics and syntax of the Flux language. Section 3 describes the Flux compiler and runtime systems. Section 4 presents our experimental methodology and compares the performance of Flux servers to their hand-written counterparts. Section 5 demonstrates the use of path profiling and discrete-event simulation. Section 6 reports our experience using Flux to build several servers. Section 7 presents related work, and Section 8 concludes with a discussion of planned future work.

2 Language Description

To introduce Flux, we develop a sample application that exercises most of Flux’s features. This sample application is an image server that receives HTTP requests for images that are stored in the PPM format and compresses them into JPEGs, using calls to an off-the-shelf JPEG library. Recently-compressed images are stored in a cache managed with a least-frequently used (LFU) replacement policy.

Figure 1 presents an abbreviated listing of the image server code with a schematic view of its dynamic execution (see Figure 2 for a more detailed listing). The `Listen` node (a “source node”) executes in an infinite loop, handling client requests and transferring data and

control to the server running the Flux program. A single Flux program represents an unbounded number of separate concurrent flows: each request executes along a separate flow through the Flux program, and eventually outputs results back to the client.

Notice that Flux programs are acyclic. The only loops exposed in Flux are the implicit infinite loops in which source nodes execute, and the round-trips between the Flux server and its clients. The lack of cycles in Flux allows it to enforce deadlock-free concurrency control. While theoretically limiting expressiveness, we have found cycles to be unnecessary for implementing in Flux the wide range of servers described in Section 4.

The Flux language consists of a minimal set of features, including **concrete nodes** that correspond to the C or C++ code implementing the server logic, **abstract nodes** that represent a flow through multiple nodes, **predicate types** that implement conditional data flow, **error handlers** that deal with exceptional conditions, and **atomicity constraints** that control simultaneous access to shared state.

2.1 Concrete Nodes

The first step in designing a Flux program is describing the *concrete nodes* that correspond to C and C++ functions. Flux requires type signatures for each node. The name of each node is followed by the input arguments in parentheses, followed by an arrow and the output arguments. Functions implementing concrete nodes require either one or two arguments. If the node is a source node, then it requires only one argument: a pointer to a struct that the function fills in with its outputs. Similarly, if the node is a sink node (without output), then its argument is a pointer to a struct that holds the function’s inputs. Most concrete nodes have two arguments: first input, then output.

Figure 2 starts with the signatures for three of the concrete nodes in the image server: `ReadRequest` parses client input, `Compress` compresses images, and `Write` outputs the compressed image to the client.

While most concrete nodes both receive input data


```

// concrete node signatures
Listen ()
=> (int socket);

ReadRequest (int socket)
=> (int socket, bool close,
    image_tag *request);

CheckCache (int socket, bool close,
    image_tag *request)
=> (int socket, bool close,
    image_tag *request);

// omitted for space:
// ReadInFromDisk, StoreInCache

Compress (int socket, bool close,
    image_tag *request,
    __u8 *rgb_data)
=> (int socket, bool close,
    image_tag *request);

Write (int socket, bool close,
    image_tag *request)
=> (int socket, bool close,
    image_tag *request);

Complete (int socket, bool close,
    image_tag *request) => ();

// source node
source Listen => Image;

// abstract node
Image = ReadRequest -> CheckCache
    -> Handler -> Write -> Complete;

// predicate type & dispatch
typedef hit TestInCache;

Handler:[_, _, hit] = ;
Handler:[_, _, _] =
    ReadInFromDisk -> Compress
    -> StoreInCache;

// error handler
handle error ReadInFromDisk => FourOhFor;

// atomicity constraints
atomic CheckCache:{cache};
atomic StoreInCache:{cache};
atomic Complete:{cache};

```

Figure 2: An image compression server, written in Flux.

and produce output, *source* nodes only produce output to initiate a data flow. The statement below indicates that

Listen is a source node, which Flux executes inside an infinite loop. Whenever Listen receives a connection, it transfers control to the Image node.

```

// source node
source Listen => Image;

```

2.2 Abstract Nodes

In Flux, concrete nodes can be composed to form *abstract nodes*. These abstract nodes represent a flow of data from concrete nodes to concrete nodes or other abstract nodes. Arrows connect nodes, and Flux checks to ensure that these connections make sense. The output type of the node on the left side of the arrow must match the input type of the node on the right side. For example, the abstract node Image in the image server corresponds to a flow from client input that checks the cache for the requested image, handles the result, writes the output, and completes.

```

// abstract node
Image = ReadRequest -> CheckCache
    -> Handler -> Write -> Complete;

```

2.3 Predicate Types

A client request for an image may result in either a cache hit or a cache miss. These need to be handled differently. Instead of exposing control flow directly, Flux lets programmers use the *predicate type* of a node's output to direct the flow of data to the appropriate subsequent node. A predicate type is an arbitrary Boolean function supplied by the Flux programmer that is applied to the node's output.

Using predicate types, a Flux programmer can express multiple possible paths for data through the server. Predicate type dispatch is processed in order of the tests in the Flux program. The typedef statement binds the type hit to the Boolean function TestInCache. The node Handler below checks to see if its first argument is of type hit; in other words, it applies the function TestInCache to the third argument. The underscores are wildcards that match any type. Handler does nothing for a hit, but if there is a miss in the cache, the image server fetches the PPM file, compresses it, and stores it in the cache.

```

// predicate type & dispatch
typedef hit TestInCache;

```

```

Handler:[_, _, hit] = ;
Handler:[_, _, _] =
    ReadInFromDisk -> Compress
    -> StoreInCache;

```

2.4 Error Handling

Any server must handle errors. Flux expects nodes to follow the standard UNIX convention of returning error codes. Whenever a node returns a non-zero value, Flux checks if an error handler has been declared for the node. If none exists, the current data flow is simply terminated.

In the image server, if the function that reads an image from disk discovers that the image does not exist, it returns an error. We handle this error by directing the flow to a node `FourOhFour` that outputs a 404 page:

```
// error handler
handle error ReadInFromDisk => FourOhFour;
```

2.5 Atomicity Constraints

All flows through the image server access a single shared image cache. Access to this shared resource must be controlled to ensure that two different data flows do not interfere with each other's operation.

The Flux programmer specifies such *atomicity constraints* in Flux rather than inside the component implementation. The programmer specifies atomicity constraints by using arbitrary symbolic names. These constraints can be thought of as locks, although this is not necessarily how they are implemented. A node only runs when it has “acquired” all of the constraints. This acquisition follows a two-phase locking protocol: the node acquires (“locks”) all of the constraints in order, executes the node, and then releases them in reverse order.

Atomicity constraints can be specified as either *readers* or *writers*. Using these constraints allows multiple readers to execute a node at the same time, supporting greater efficiency when most nodes read shared data rather than update it. Reader constraints have a question mark appended to them (“?”). Although constraints are considered writers by default, a programmer can append an exclamation point (“!”) for added documentation.

In the image server, the image compression cache can be updated by three nodes: `CheckCache`, which increments a reference count to the cached item, `StoreInCache`, which writes a new item into the cache, evicting the least-frequently used item with a zero reference count, and `Complete`, which decrements the cached image's reference count. Only one instance of each node may safely execute at a time; since all of them modify the cache, we label them with the same writer constraint (`cache`).

```
// atomicity constraints
atomic CheckCache: {cache};
atomic StoreInCache: {cache};
atomic Complete: {cache};
```

Note that programmers can apply atomicity constraints not only to concrete nodes but also to abstract

nodes. In this way, programmers can specify that multiple nodes must be executed atomically. For example, the node `Handler` could also be annotated with an atomicity constraint, which would span the execution of the path `ReadInFromDisk` → `Compress` → `StoreInCache`. This freedom to apply atomicity constraints presents some complications for deadlock-free lock assignment, which we discuss in Section 3.1.1.

2.5.1 Scoped Constraints

While flows generally represent independent clients, in some server applications, multiple flows may constitute a single *session*. For example, a file transfer to one client may take the form of multiple simultaneous flows. In this case, the state of the session (such as the status of transferred chunks) only needs to be protected from concurrent access in that session.

In addition to *program-wide constraints* that apply across the entire server (the default), Flux supports *per-session constraints* that apply only to particular sessions. Using session-scoped atomicity constraints increases concurrency by eliminating contention across sessions. Sessions are implemented as hash functions on the output of each source node. The Flux programmer implements a session id function that takes the source node's output as its parameter and returns a unique session identifier, and then adds (`session`) to a constraint name to indicate that it applies only per-session.

2.5.2 Discussion

Specifying atomicity constraints in Flux rather than placing locking operations inside implementation code has a number of advantages, beyond the fact that it allows the use of libraries whose source code is unavailable.

Safety. The Flux compiler imposes a canonical ordering on atomicity constraints (see Section 3.1.1). Combined with the fact that Flux flows are acyclic, this ordering prevents cycles from appearing in its lock graph. Programs that use Flux-level atomicity constraints exclusively (i.e., that do not themselves contain locking operations) are thus guaranteed to not deadlock.

Efficiency. Exposing atomicity constraints also enables the Flux compiler to generate more efficient code for particular environments. For example, while a multi-threaded runtime requires locks, a single-threaded event-driven runtime does not. The Flux compiler generates locks or other mutual exclusion operations only when needed.

Granularity selection. Finally, atomicity constraints let programmers easily find the appropriate granularity of locking — they can apply fine-grained constraints to individual concrete nodes or coarse-grained constraints to abstract nodes that comprise many concrete nodes.

However, even when deadlock-freedom is guaranteed, grain selection can be difficult: too coarse a grain results in contention, while too fine a grain can impose excessive locking overhead. As we describe in Section 5.1, Flux can generate a discrete event simulator for the Flux program. This simulator can let a developer measure the effect of different granularity decisions and identify the appropriate locking granularity before actual server deployment.

3 Compiler and Runtime Systems

A Flux program is transformed into a working server by a multi-stage process. The compiler first reads in the Flux source and constructs a representation of the program graph. It then processes the internal representation to type-check the program. Once the code has been verified, the runtime code generator processes the graph and outputs C code that implements the server's data flow for a specific runtime. Finally, this code is linked with the implementation of the server logic into an operational server. We first describe the compilation process in detail. We then describe the three runtime systems that Flux currently supports.

3.1 The Flux Compiler

The Flux compiler is a three-pass compiler implemented in Java, and uses the JLex lexer [5] in conjunction with the CUP LALR parser generator [3].

The first pass parses the Flux program text and builds a graph-based internal representation. During this pass, the compiler links nodes referenced in the program's data flows. All of the conditional flows are merged, with an edge corresponding to each conditional flow.

The second pass decorates edges with types, connects error handlers to their respective nodes, and verifies that the program is correct. First, each node mentioned in a data flow is labelled with its input and output types. Each predicate type used by a conditional node is associated with its user-supplied predicate function. Finally, the error handlers and atomicity constraints are attached to each node. If any of the referenced nodes or predicate types are undefined, the compiler signals an error and exits. Otherwise, the program graph is completely instantiated. The final step of program graph construction checks that the output types of each node match the inputs of the nodes that they are connected to. If all type tests pass, then the compiler has a valid program graph.

The third pass generates the intermediate code that implements the data flow of the server. Flux supports generating code for arbitrary runtime systems. The compiler defines an object-oriented interface for code generation. New runtimes can easily be plugged into the Flux compiler by implementing this code generator interface.

The current Flux compiler supports several different

runtimes, described below. In addition to the runtime-specific intermediate code, the Flux compiler generates a Makefile and stubs for all of the functions that provide the server logic. These stubs ensure that the programmer uses the appropriate signatures for these methods. When appropriate, the code generator outputs locks corresponding to the atomicity constraints.

3.1.1 Avoiding Deadlock

The Flux compiler generates locks in a canonical order. Our current implementation sorts them alphabetically by name. In other words, a node that has y, x as its atomicity constraints actually first acquires x , then y .

When applied only to concrete nodes, this approach straightforwardly combines with Flux's acyclic graphs to eliminate deadlock. However, when abstract nodes also require constraints, the constraints may become nested and preventing deadlock becomes more complicated. Nesting could itself cause deadlock by acquiring constraints in non-canonical order. Consider the following Flux program fragment:

```
A = B;
C = D;

atomic A: {x};
atomic B: {y};
atomic C: {y};
atomic D: {x};
```

In this example, a flow passing through A (which then invokes B) locks x and then y . However, a flow through C locks y and then x , which is a cycle in the locking graph.

To prevent deadlock, the Flux compiler detects such situations and moves up the atomicity constraints in the program, forcing earlier lock acquisition. For each abstract node with atomicity constraints, the Flux compiler computes a constraint list comprising the atomicity constraints the node transitively requires, in execution order. This list can easily be computed via a depth-first traversal of the relevant part of the program graph. If a constraint list is out of order, then the first constraint acquired in a non-canonical order is added to the parent of the node that requires the constraint. This process repeats until no out-of-order constraint lists remain.

For the above example, Flux will discover that node C has an out-of-order sequence (y, x). It then adds constraint x to node C. The algorithm then terminates with the following set of constraints:

```
atomic A: {x};
atomic B: {y};
atomic C: {x, y};
atomic D: {x};
```

Flux locks are reentrant, so multiple lock acquisitions do not present any problems. However, reader and writer locks require special treatment. After computing all constraint lists, the compiler performs a second pass to find any instances when a lock is acquired at least once as a reader and a writer. If it finds such a case, Flux changes the first acquisition of the lock to a writer if it is not one already. Reacquiring a constraint as a reader while possessing it as a writer is allowed because it does not cause the flow to give up the writer lock.

Because early lock acquisition can reduce concurrency, whenever the Flux compiler discovers and resolves potential deadlocks as described above, it generates a warning message.

3.2 Runtime Systems

The current Flux compiler supports three different runtime systems: one thread per connection, a thread-pool system, and an event-driven runtime.

3.2.1 Thread-based Runtimes

In the thread-based runtimes, each request handled by the server is dispatched to a thread function that handles all possible paths through the server's data flows. In the one-to-one thread server, a thread is created for every different data flow. In the thread-pool runtime, a fixed number of threads are allocated to service data flows. If all threads are occupied when a new data flow is created, the data flow is queued and handled in first-in first-out order.

3.2.2 Event-driven Runtime

Event-driven systems can provide robust performance with lower space overhead than thread-based systems [25]. In the event-driven runtime, every input to a functional node is treated as an event. Each event is placed into a queue and handled in turn by a single thread. Additionally, each source node (a node with no input) is repeatedly added to the queue to originate each new data flow. The transformation of input to output by a node generates a new event corresponding to the output data propagated to the subsequent node.

The implementation of the event-based runtime is complicated by the fact that node implementations may perform blocking function calls. If blocking function calls like `read` and `write` were allowed to run unmodified, the operation of the entire server would block until the function returned.

Instead, the event-based runtime intercepts all calls to blocking functions using a handler that is pre-loaded via the `LD_PRELOAD` environment variable. This handler captures the state of the node at the blocking call and moves to the next event in the queue. The formerly-blocking call is then executed asynchronously. When

the event-based runtime receives a signal that the call has completed, the event is reactivated and re-queued for completion. Because the mainstream Linux kernel does not currently support callback-driven asynchronous I/O, the current Flux event-based runtime uses a separate thread to simulate callbacks for asynchronous I/O using the `select` function. A programmer is thus free to use synchronous I/O primitives without interfering with the operation of the event-based runtime.

3.2.3 Other Languages and Runtimes

Each of these runtimes was implemented in the C using POSIX threads and locks. Flux can also generate code for different programming languages. We have also implemented a prototype that targets Java, using both SEDA [25] and a custom runtime implementation, though we do not evaluate the Java systems here.

In addition to these runtimes, we have implemented a code generator that transforms a Flux program graph into code for the discrete event simulator CSIM [18]. This simulator can predict the performance of the server under varying conditions, even prior to the implementation of the core server logic. Section 5.1 describes this process in greater detail.

4 Experimental Evaluation

To demonstrate its effectiveness for building high-performance server applications, we implemented a number of servers in Flux. We summarize these in Table 1. We chose these servers specifically to span the space of possible server applications. Most server applications can be broadly classified into one of the following categories, based on how they interact with clients: request-response client/server, "heartbeat" client/server and peer-to-peer.

We implemented a server in Flux for each of these categories and compared its performance under load with existing hand-tuned server applications written in conventional programming languages. The Flux servers rely on single-threaded C and C++ code that we either borrowed from existing implementations or wrote ourselves. The most significant inclusions of existing code were in the web server, which uses the PHP interpreter, and in the image server, which relies on calls to the `libjpeg` library to compress JPEG images.

4.1 Methodology

We evaluate all server applications by measuring their throughput and latency in response to realistic workloads.

All testing was performed with a server and client machine, both running Linux version 2.4.20. The server machine was a Pentium 4 (2.4Ghz, 1GB RAM), connected via gigabit Ethernet on a dedicated switched

Server	Style	Description	Lines of Flux code	Lines of C/C++ code
Web server	request-response	a basic HTTP/1.1 server with PHP	36	386 (+ PHP)
Image server	request-response	image compression server	23	551 (+ libjpeg)
BitTorrent	peer-to-peer	a file-sharing server	84	878
Game server	heartbeat client-server	multiplayer game of “Tag”	54	257

Table 1: Servers implemented using Flux, described in Section 4.

network to the client machine, a Xeon-based machine (2.4Ghz, 1GB RAM). All server and client applications were compiled using GCC version 3.2.2. During testing, both machines were running in multi-user mode with only standard services running. All results are for a run of two minutes, ignoring the first twenty seconds to allow the cache to warm up.

4.2 Request-Response: Web Server

Request-response based client/server applications are among the most common examples of network servers. This style of server includes most major Internet protocols including FTP, SMTP, POP, IMAP and HTTP. As an example of this application class, we implemented a web server in Flux. The Flux web server implements the HTTP/1.1 protocol and can serve both static and dynamic PHP web pages.

We implemented a benchmark to load test the Flux webserver that is similar to SPECweb99 [21]. The benchmark simulates a number of clients requesting files from the server. Each simulated client sends five requests over a single HTTP/1.1 TCP connection using keep-alives. When one file is retrieved, the next file is immediately requested. After the five files are retrieved, the client disconnects and reconnects over a new TCP connection. The files requested by each simulated client follow the static portion of the SPECweb benchmark and each file is selected using the Zipf distribution. The working set for this benchmark is approximately 32MB, which fits into RAM, so this benchmark primarily stresses CPU performance.

We compare the performance of the Flux webserver against the latest versions of the *knot* webserver distributed with Capriccio [24] and the *Haboob* webserver distributed with the SEDA runtime system [25]. Figure 3 presents the throughput and latency for a range of simultaneous clients. These graphs represent the average of five different runs for each number of clients.

The results show that the Flux web server provides comparable performance to the fastest webserver (*knot*), regardless of whether the event-based or thread-based runtime is used. All three of these servers (*knot*, *flux-threadpool* and *flux-event-based*) significantly outperform *Haboob*, the event-based server distributed with SEDA. As expected, the naïve one-thread, one-client

server generated by Flux has significantly worse performance due to the overhead of creating and destroying threads.

The results for the event-based server highlight one drawback of running on a system without true asynchronous I/O. With small numbers of clients, the event-based server suffers from increased latency that initially decreases and then follows the behavior of the other servers. This hiccup is an artifact of the interaction between the webserver’s implementation and the event-driven runtime, which must simulate asynchronous I/O. The first node in the webserver uses the `select` function with a timeout to wait for network activity. In the absence of other network activity, this node will block for a relatively long period of time. Because the event-based runtime only reactivates nodes that make blocking I/O calls after the completion of the currently-operating node, in the absence of other network activity, the call to `select` imposes a minimum latency on all blocking I/O. As the number of clients increases, there is sufficient network activity that `select` never reaches its timeout and frozen nodes are reactivated at the appropriate time. In the absence of true asynchronous I/O, the only solution to this problem would be to decrease the timeout call to `select`, which would increase the CPU usage of an otherwise idle server.

4.3 Peer-to-Peer: BitTorrent

Peer-to-peer applications act as both a server and a client. Unlike a request-response server, they both receive and initiate requests.

We implemented a BitTorrent server in Flux as a representative peer-to-peer application. BitTorrent uses a scatter-gather protocol for file sharing. BitTorrent peers exchange pieces of a shared file until all participants have a complete copy. Network load is balanced by randomly requesting different pieces of the file from different peers.

To facilitate benchmarking, we changed the behavior of both of the BitTorrent peers we test here (the Flux version and CTorrent). First, all client peers are *unchoked* by default. Choking is an internal BitTorrent state that blocks certain clients from downloading data. This protocol restriction prevents real-world servers from being overwhelmed by too many client requests. We also allow

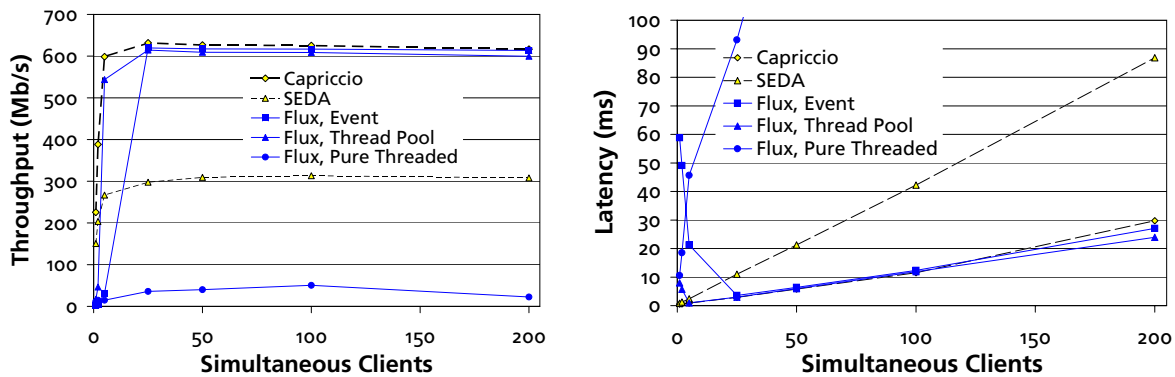


Figure 3: Comparison of Flux web servers with other high-performance implementations (see Section 4.2).

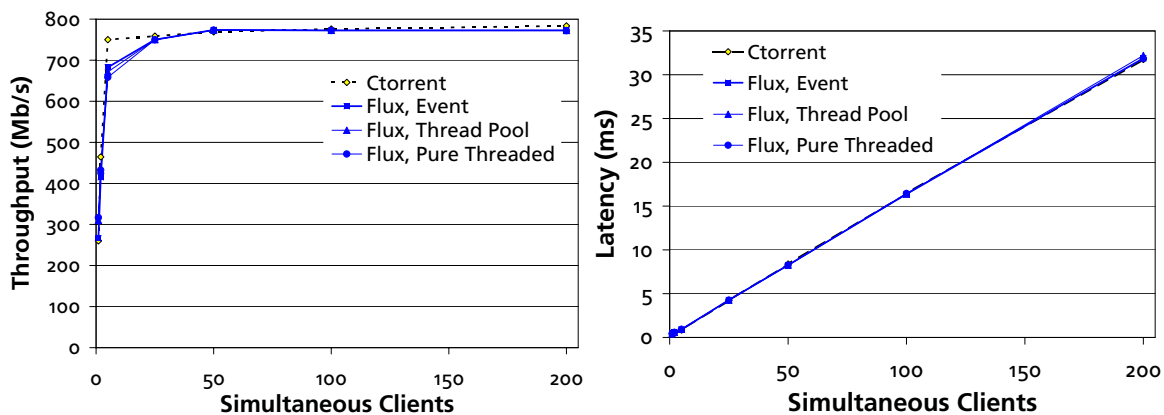


Figure 4: Comparison of Flux BitTorrent servers with CTorrent (see Section 4.3).

an unlimited number of unchoked client peers to operate simultaneously, while the real BitTorrent server only unchokes clients who upload content.

We are unaware of any existing BitTorrent benchmarks, so we developed our own. Our BitTorrent benchmark mimics the traffic encountered by a busy BitTorrent peer and stresses server performance. It simulates a series of clients continuously sending requests for randomly distributed pieces of a 54MB test file to a BitTorrent peer with a complete copy of the file. When a peer finishes downloading a piece of the file, it immediately requests another random piece of the file from those still missing. Once a client has obtained the entire file, it disconnects. This benchmark does not simulate the “scatter-gather” nature of the BitTorrent protocol; instead, all requests go to a single peer. Using single peers has the effect of maximizing load, since obtaining data from a different source would lessen the load on the peer being tested.

Figure 4 compares the latency, throughput in completions per second and network throughput to CTorrent, an implementation of the BitTorrent protocol written in C. The goal of any BitTorrent system is to maximize

network utilization (thus saturating the network), and both the CTorrent and Flux implementations achieve this goal. However, prior to saturating the network, all of the Flux servers perform slightly worse than the CTorrent server. We are investigating the cause of this small performance gap.

4.4 Heartbeat Client-Server: Game Server

Unlike request-response client/server applications and most peer-to-peer applications, certain server applications are subject to deadlines. An example of such a server is an online multi-player game. In these applications, the server maintains the shared state of the game and distributes this state to all of the players at “heartbeat” intervals. There are two important conditions that must be met by this communication: the state possessed by all clients must be the same at each instant in time, and the inter-arrival time between states can not be too great. If either of these conditions is violated, the game will be unplayable or susceptible to cheating. These requirements place an important delay-sensitive constraint on the server’s performance.

We have implemented an online multi-player game of Tag in Flux. The Flux game server enforces the rules of Tag. Players can not move beyond the boundaries of the game world. When a player is tagged by the player who is “it”, that player becomes the new “it” and is teleported to a new random location on the board. All communication between clients and server occurs over UDP at 10Hz, a rate comparable to other real-world online games. While simple, this game has all of the important characteristics of servers for first person shooter or real-time strategy games.

Benchmarking the gameserver is significantly different than load-testing either the webserver or BitTorrent peer. Throughput is not a consideration since only small pieces of data are transmitted. The primary concern is the latency of the server as the number of clients increases. The server must receive every player’s move, compute the new game state, and broadcast it within a fixed window of time.

To load-test the game server, we measured the effect of increasing the number of players. The performance of the gameserver is largely based upon the length of time it takes the server to update the game state given the moves received from all of the players, and this computation time is identical across the servers. The latency of the gameserver is largely a product of the rate of game turns, which stays constant at 10Hz. We found no appreciable differences between a traditional implementation of the gameserver and the various Flux versions. These results show that Flux is capable of producing a server with sufficient performance for multi-player online gaming.

5 Performance

In addition to its programming language support for writing server applications, Flux provides support for predicting and measuring the performance of server applications. The Flux system can generate **discrete-event simulators** that predict server performance for synthetic workloads and on different hardware. It can also perform **path profiling** to identify server performance bottlenecks on a deployed system.

5.1 Performance Prediction

Predicting the performance of a server prior to deployment is important but often difficult. For example, performance bottlenecks due to contention may not appear during testing because the load placed on the system is insufficient. In addition, system testing on a small-scale system may not reveal problems that arise when the system is deployed on an enterprise-scale multiprocessor.

In addition to generating executable server code, the Flux code generator can automatically transform a Flux program directly into a discrete-event simulator that

```
void Image () {
    rw_write_lock(lock);
    processor->reserve();
    hold(exponential(CMP_TIME_CPU_IMAGE));
    processor->release();
    rw_write_unlock(lock);
    // Call the next Node
    ReadRequest();
}
```

Figure 5: Compiler-generated discrete-event simulation code for a Flux node.

models the performance of the server. The implementation language for the simulator is CSim [18], a C-based, process-oriented simulator.

In the simulator, CPUs are modeled as resources that each Flux node acquires for a given amount of time. The simulator can either use observed parameters from a running system (per-node execution times, source node inter-arrival times, and observed branching probabilities), or the Flux programmer can supply estimates for these parameters. The simulator can model an arbitrary number of processors by increasing the number of nodes that may simultaneously acquire the CPU resource. When a node uses a given atomicity constraint, it treats it as a lock and acquires it for the duration of the node’s execution. While the simulator accurately models both reader and writer constraints, it conservatively treats session-level constraints as globals.

The code in Figure 5 is a simplified version of the CSim code that Flux generates. Here, the node Image has a writer lock associated with it. Once CSim schedules this node onto a CPU, it models its execution time using an exponential distribution based on the observed CPU time collected from a profiling run. Finally, this node unlocks its reader-writer lock and executes the next node in the flow.

It is important to note that this simulation does not model disk or network resources. While this is a realistic assumption for CPU-bound servers (such as dynamic web-servers), other servers may require more complete modeling.

To demonstrate that the generated simulations accurately predict actual performance, we tested the image server described in Section 2. To simulate load on the machine, we made requests at increasingly small inter-arrival times. The image server had 5 images, and our load tester randomly requests one of eight sizes (between 1/8th scale and full-size) of a randomly-chosen image. When configured to run with n “clients”, the load tester issues requests at a rate of one every $1/n$ seconds. The image server is CPU-bound, with each image taking on average 0.5 seconds to compress.

We first measured the performance of this server on a 16-processor SunFire 6800, but with only a single CPU enabled. We then used the observed node runtime and branching probabilities to parameterize the generated CSIM simulator. We compare the predicted and actual performance of the server by making more processors available to the system. As Figure 6 shows, the predicted results (dotted lines) and actual results (solid lines) match closely, demonstrating the effectiveness of the simulator at predicting performance.

5.2 Path Profiling

The Flux compiler optionally instruments generated servers to simplify the identification of performance bottlenecks. This profiling information takes the form of “hot paths”, the most frequent or most time-consuming paths in the server. Flux identifies these hot paths using the Ball-Larus path profiling algorithm [4]. Because Flux graphs are acyclic, the Ball-Larus algorithm identifies each unique path through the server’s data-flow graph.

Hot paths not only aid understanding of server performance characteristics but also identify places where optimization would be most effective. Because profiling information can be obtained from an operating server and is linked directly to paths in the program graph, a performance analyst can easily understand the performance characteristics of deployed servers.

The overhead of path profiling is low enough that hot path information can be maintained even in a production server. Profiling adds just one arithmetic operation and two high-resolution timer calls to each node. A performance analyst can obtain path profiles from a running Flux server by connecting to a dedicated socket.

To demonstrate the use of path profiling, we compiled a version of the BitTorrent peer with profiling enabled. For the experiments, we used a patched version of Linux that supports per-thread time gathering. The BitTorrent peer was load-tested with the same tester as in the performance experiments. For profiling, we used loads of 25, 50, and 100 clients. All profiling information was automatically generated from a running Flux server.

In BitTorrent, the most time-consuming path identified by Flux was, unsurprisingly, the file transfer path (Listen → GetClients → SelectSockets → CheckSockets → Message → ReadMessage → HandleMessage → Request → MessageDone, 0.295 ms). However, the second most expensive path was the path that finds no outstanding chunk requests (Listen → GetClients → SelectSockets → CheckSockets → ERROR, 0.016ms). While this path is relatively cheap compared to the file transfer path, it also turns out to be the most frequently executed path (780,510 times, compared to 313,994 for the file transfer

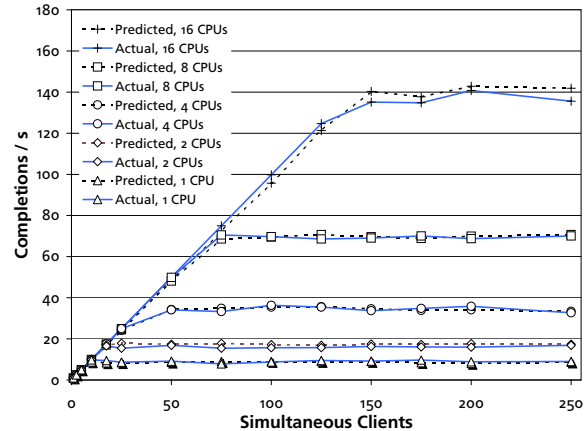


Figure 6: Predicted performance of the image server (derived from a single-processor run) versus observed performance for varying numbers of processors and load.

path). Since this path accounts for 13% of BitTorrent’s execution time, it is a reasonable candidate for optimization efforts.

6 Developer Experience

In this section, we examine the experience of programmers implementing Flux applications. In particular, we focus on the implementation of the Flux BitTorrent peer.

The Flux BitTorrent peer was implemented by two undergraduate students in less than one week. The students began with no knowledge of the technical details of the BitTorrent protocol or the Flux language. The design of the Flux program for the BitTorrent peer was entirely their original work. The implementation of the functional nodes in BitTorrent is loosely derived from the CTorrent source code. The program graph for the BitTorrent server is shown in Figure 7 at the end of this document.

The students had a generally positive reaction to programming in Flux. Primarily, they felt that organizing the application into a Flux program graph prior to implementation helped modularize their application design and debug server data flow prior to programming. They also found that the exposure of atomicity constraints at the Flux language level allowed for easy identification of the appropriate locations for mutual exclusion. Flux’s immunity to deadlock and the simplicity of the atomicity constraints increased their confidence in the correctness of the resulting server.

Though this is only anecdotal evidence, this experience suggests that programmers can quickly gain enough expertise in Flux to build reasonably complex server applications.

7 Related Work

This section discusses related work to Flux in the areas of coordination and data flow languages, programming language constructs, domain-specific languages, and runtime systems.

Coordination and data flow languages. Flux is an example of a *coordination language* [11] that combines existing code into a larger program in a data flow setting. There have been numerous data flow languages proposed in the literature, see Johnston et al. for a recent survey [13]. Data flow languages generally operate at the level of fundamental operations rather than at a functional granularity. One exception is CODE 2, which permits incorporation of sequential code into a dynamic flow graph, but restricts shared state to a special node type [7, 8]. Data flow languages also typically prohibit global state. For example, languages that support *streaming* applications like StreamIt [22] express all data dependencies in the data flow graph. Flux departs from these languages by supporting safe access to global state via atomicity constraints. Most importantly, these languages focus on extracting parallelism from individual programs, while Flux describes parallelism across multiple clients or event streams.

Programming language constructs. Flux shares certain linguistic concepts with previous and current work in other programming languages. Flux's predicate matching syntax is deliberately based on the pattern-matching syntax used by functional languages like ML, Miranda, and Haskell [12, 19, 23]. The PADS data description language also allows programmers to specify predicate types, although these must be written in PADS itself rather than in an external language like C [9]. Flanagan and Freund present a type inference system that computes "atomicity constraints" for Java programs that correspond to Lipton's theory of reduction [10, 16]; Flux's atomicity constraints operate at a higher level of abstraction. The Autolocker tool [17], developed independently and concurrently with this work, automatically assigns locks in a deadlock-free manner to manually-annotated C programs. It shares Flux's enforcement of an acyclic locking order and its use of two-phase lock acquisition and release.

Related domain-specific languages. Several previous domain-specific languages allow the integration of off-the-shelf code into data flow graphs, though for different domains. The Click modular router is a domain-specific language for building network routers out of existing C components [14]. Knit is a domain-specific language for building operating systems, with rich support for integrating code implementing COM interfaces [20]. In addition to its linguistic and tool support for programming server applications, Flux ensures deadlock-freedom by enforcing a canonical lock ordering; this is

not possible in Click and Knit because they permit cyclic program graphs.

Runtime systems. Researchers have proposed a wide variety of runtime systems for high-concurrency applications, including SEDA [25], Hood [1, 6], Capriccio [24], Fibers [2], cohort scheduling [15], and liba-sync/mp [26], whose per-callback *colors* could be used to implement Flux's atomicity constraints. Users of these runtimes are forced to implement a server using a particular API. Once implemented, the server logic is generally inextricably linked to the runtime. By contrast, Flux programs are independent of any particular choice of runtime system, so advanced runtime systems can be integrated directly into Flux's code generation pass.

8 Future Work

We plan to build on this work in several directions. First, we are actively porting Flux to other architectures, especially multicore systems. We are also planning to extend Flux to operate on clusters. Because concurrency constraints identify nodes that share state, we plan to use these constraints to guide the placement of nodes across a cluster to minimize communication.

To gain more experience with Flux, we are adding further functionality to the web server. In particular, we plan to build an Apache compatibility layer so we can easily incorporate Apache modules. We also plan to enhance the simulator framework to support per-session constraints.

The entire Flux system is available for download at `flux.cs.umass.edu` via the Flux-based BitTorrent and web servers described in this paper.

9 Acknowledgments

The authors thank Gene Novark for helping to design the discrete event simulation generator, and Vitaliy Lvin for assisting in experimental setup and data gathering.

This material is based upon work supported by the National Science Foundation under CAREER Awards CNS-0347339 and CNS-0447877. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

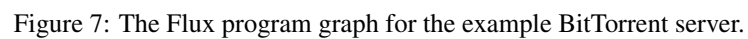
References

- [1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM Press.
- [2] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management

- without manual stack management. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.
- [3] A. W. Appel, F. Flannery, and S. E. Hudson. CUP parser generator for Java. <http://www.cs.princeton.edu/~appel/modern/java/CUP/>.
 - [4] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.
 - [5] E. Berk and C. S. Ananian. JLex: A lexical analyzer generator for Java. <http://www.cs.princeton.edu/~appel/modern/java/JLex/>.
 - [6] R. D. Blumofe and D. Papadopoulos. The performance of work stealing in multiprogrammed environments (extended abstract). In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 266–267, New York, NY, USA, 1998. ACM Press.
 - [7] J. C. Browne, M. Azam, and S. Sobek. CODE: A unified approach to parallel programming. *IEEE Software*, 6(4):10–18, 1989.
 - [8] J. C. Browne, E. D. Berger, and A. Dube. Compositional development of performance models in POEMS. *The International Journal of High Performance Computing Applications*, 14(4):283–291, Winter 2000.
 - [9] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 295–304, New York, NY, USA, 2005. ACM Press.
 - [10] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 47–58, New York, NY, USA, 2005. ACM Press.
 - [11] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96, 1992.
 - [12] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.
 - [13] W. M. Johnston, J. R. P. Hanna, and R. J. Milner. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.
 - [14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.
 - [15] J. R. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 103–114, Berkeley, CA, USA, 2002. USENIX Association.
 - [16] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
 - [17] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In J. G. Morrisett and S. L. P. Jones, editors, *POPL*, pages 346–358. ACM, Jan. 2006.
 - [18] Mesquite Software. The CSIM Simulator. <http://www.mesquite.com>.
 - [19] R. Milner. A proposal for standard ml. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 184–197, New York, NY, USA, 1984. ACM Press.
 - [20] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of the 4th ACM Symposium on Operating Systems Design and Implementation (OSDI)*, pages 347–360, Oct. 2000.
 - [21] Standard Performance Evaluation Corporation. SPECweb99. <http://www.spec.org/osg/web99/>.
 - [22] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.
 - [23] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
 - [24] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM Press.
 - [25] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the*

eighteenth ACM symposium on Operating systems principles, pages 230–243, New York, NY, USA, 2001. ACM Press.

- [26] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.



Understanding and Addressing Blocking-Induced Network Server Latency

Yaoping Ruan
IBM T.J. Watson Research Center
Yorktown Heights, NY
{yaoping.ruan}@us.ibm.com

Vivek Pai
Department of Computer Science
Princeton University
{vivek}@cs.princeton.edu

Abstract

We investigate the origin and components of network server latency under various loads and find that filesystem-related kernel queues exhibit head-of-line blocking, which leads to bursty behavior in event delivery and process scheduling. In turn, these problems degrade the existing fairness and scheduling policies in the operating system, causing requests that could have been served in memory, with low latency, to unnecessarily wait on disk-bound requests. While this batching behavior only mildly affects throughput, it severely degrades latency. This problem manifests itself in fairness and service quality degradation, a phenomenon we call *service inversion*.

We show a portable solution that avoids these problems without kernel or filesystem modifications. We modify two different Web servers to use this approach, and demonstrate a qualitatively different change in their latency profiles, generating more than an order of magnitude reduction in latency. The resulting systems are able to serve most requests without being tied to disk performance, and they scale better with improvements in processor speed. These results are not dependent on server software architecture, and can be profitably applied to experimental and production servers.

1 Introduction

Much of the performance-related research in network servers has focused on improving throughput, with less attention paid to latency [6, 13]. In an environment with large numbers of users accessing the Web over slow links, the focus on throughput was understandable, since perceived latency was dominated by wide area network (WAN) delays. Additionally, early servers were often unable to handle high request rates, so throughput research directly affected service availability. The development of popular throughput-centric benchmarks, such as SPECWeb96 [19] and WebStone [12], also gave developers extra incentive to improve throughput.

Several trends are reducing the non-server latencies, thereby increasing the relative contribution of server-induced latency. Improvements in server-side network

connectivity reduce server-side network delays, while growing broadband usage reduces client-side network delays. Content distribution networks, which replicate content geographically, reduce the distance between the client and the desired data, reducing round-trip latency. With latencies between most major cities in the mainland US on the order of tens of milliseconds, server induced latency could be a significant portion of end-user perceived latency. Some recent work addresses the issue of measuring end-user latency [3, 15], with optimization approaches mostly focusing on scheduling [5, 9, 20, 21].

However, comparatively little is understood about trends in network server latencies, or how system components affect them. Current research generally assumes that server latency is largely caused by queuing delays, that it is inherent to the system, and that scheduling techniques are the preferred solution to address them. Unfortunately, these assumptions are not explicitly tested, complicating attempts to systematically address issues of latency. Based on these observations, our goal is to understand the root causes of network server latency and address them, so that server latency can be improved. A better understanding of latency's origins can also help other research, such as improving Quality-of-Service (QoS) or scheduling policies.

By instrumenting the kernel, we find that Web servers can incur latency blocked in filesystem-related system calls, even when the needed data is in physical memory. As a result, requests that could have been served from main memory are forced to wait unnecessarily for disk-bound requests. This batching behavior may have little impact on throughput, it can significantly affect latency. It causes head-of-line blocking in the OS and manifests itself as other problems, such as a degradation of the kernel's service policies that are designed to ensure fairness. By examining individual request latencies, we find that this blocking reduces the fairness of response orders, a phenomenon we call *service inversion*, where short requests are often served with much higher latencies than much larger requests. We also find that this phenomenon increases with load, and that it is responsible for most of the growth in server latency under load.

By addressing the blocking issues both in the application and the kernel, we improve response time by more than an order of magnitude, and demonstrate qualitatively different change in the latency profiles. The resulting servers also exhibit much lower service inversion and better fairness. These latency profiles in our resulting servers generally scale with processor speed, where cached requests are no longer bound by disk-related issues. In comparison, experiments using the original servers only show that server throughput improves with increases in processor speed, but not server latency. We believe that our solution is more portable than redesigning kernel locking, and that our findings also apply to Web proxies, where more disk activity is required and the working sets generally exceed physical memory.

The rest of the paper is organized as follow: In Section 2, we present the servers used throughout this paper, test environment, workloads, and methodology. In Section 3 we identify the latency problems and explain their causes. We introduce a new metric to quantify the effects in Section 4. In Section 5, we discuss how we address these problems, describe the resulting servers, present the experimental results on the new servers, and examine latency scalability with processor speeds. We discuss related work in Section 6 and conclude in Section 7.

2 Background

In this section we provide some background on our previous work, and describe the network servers, experimental setup, workloads and methodology since we begin our analysis with experimental measurements of the servers. Our earlier work on performance debugging tools [17] examined blocking in servers, but did not specifically try to understand the origins of latency, our main contribution in this work.

2.1 Server Software

To test the common scenario as well as a more aggressive case, we use two different servers with different software architectures and design goals. To represent widely-deployed general-purpose servers, we use the multi-process Apache server [1], version 1.3.27. To test high-performance servers, we use the event-driven Flash Web Server [13], a research system with aggressive optimizations. Where appropriate, we test two versions of Flash – one using the standard `select()` system call for event delivery, as well as one that uses the more scalable `kevent()` event-delivery mechanism coupled with the zero-copy `sendfile()` system call.

The Apache server utilizes blocking system calls and relies on the operating system's scheduling policy to provide parallelism, while Flash uses event delivery mechanism to multiplex all client connections. Flash consists of

Processor	P-II	P-III	P4 Xeon
Speed (MHz)	300	933	3000
Bcopy bandwidth (MB/s)	93	265	624
Read bandwidth (MB/s)	213	555	1972
Memory latency (ns)	245	101	116

Table 1: Server hardware information – hardware characteristics of three generations of the Intel Pentium processor, with some values measured by `lmbench` [11]

a single main process using non-blocking sockets, and a small set of helper processes performing disk-related operations. To increase performance, it aggressively caches open files, memory-mapped data, and application-level metadata. In contrast, Apache dedicates one process per connection, and performs very little caching in order to reduce resource consumption.

In our experiments, both servers are configured for maximum performance. In Flash, the file cache size is set to 80% of the physical memory, with remaining parameters automatically adjusted. We also aggressively configure Apache – periodic process shutdown is disabled, reverse lookups are disabled, the maximum number of processes is raised to 2048, and access logging is disabled in both servers.

2.2 Experimental Setup

Our main test platform is a uniprocessor 3.06GHz Pentium-4 with 1GB physical memory, one 5600 RPM Maxtor IDE disk, and a single Netgear GA621 gigabit Ethernet network adaptor. We use six 1.3 GHz AMD Duron machines as clients, with 256 MB of memory per machine. The network is a Netgear FS518 Gigabit Ethernet switch. All machines are configured to use the default (1500 byte) MTU. We use the FreeBSD 4.6 operating system, with all tunable parameters set for high performance – 128K max sockets, 64K file descriptors per process, 64KB socket buffers, 80K mbufs, 40K mbuf clusters, and 16K inode cache entries. We also investigate latency scalability using three hardware platforms which span three processor generations and an order of magnitude increase in raw clock speed. To equalize as many factors as possible, all machines use the same disk and network interface. The details of our server machines are shown in Table 1, with measured values provided by `lmbench` [11].

2.3 Workloads

In order to use a widely-understood workload while still maintaining tractability in the analysis, we focus on a static content workload modeled on the SPECWeb96 and SPECWeb99 [19] benchmarks. These workloads are modeled after the access patterns of multiple Web sites,

Data Set Size	Top 50 %	Top 90 %	Top 95 %	Top 99 %
1024	2.1	39.5	64.6	138.3
2048	3.0	72.9	123.6	262.8
3072	4.4	101.8	181.2	385.7
4096	4.9	131.8	235.0	505.0

Table 2: SPECWeb’s popularity distributions. All sizes shown in MB. Sizes do not scale linearly with the total data set size because directories are weighted using a Zipf popularity distribution

with file sizes ranging from 100 bytes to 900 KB, and are the *de facto* standards in industry, with more than 200 published results. File popularity is explicitly modeled – half of all accesses are for files in the 1KB-9KB range, with 35% in the 100-900 byte range, 14% in the 10KB-90KB range, and 1% in the 100KB-900KB range, yielding an average dynamic response size of roughly 14 KB. Each directory in the system contains 36 files (roughly 5 MB total), and the directories are chosen using a Zipf distribution with an alpha value of 1. The strong bias toward small files leads to the result that the most popular files consume very little aggregate space. Table 2 illustrates this heavy-tail feature well – the most popular 99% of the requests occupy at most 14% of the size of data set.

SPECWeb normally self-scales, increasing both data set size and number of simultaneous connections with the target throughput. However, this approach complicates comparisons between different servers, so we use fixed values for both parameters. To facilitate comparisons with previous work such as Haboob [21] and Knot [20], we use their parameters of a 3GB data set and 1024 simultaneous connections. With this data set size, most requests can be served from memory while a small portion will cause disk access. We also adopt the persistent connection model from these tests, with clients issuing 5 requests per connection before closing it. With these parameters, we maintain per-client throughput levels comparable to SPECWeb99’s quality-of-service requirements.

2.4 Measurement Methodology

To understand how load affects response time, we measure latencies at various requests rates. Each server’s maximum capacity is determined by having all clients issue requests in an infinite-demand (saturation) model, which is defined as load level of 1, and then relative rates are reported as load fractions relative to the infinite demand capacity of each server. This process simplifies comparison across servers, though it may bias toward servers with low capacity. Response time is measured by recording the wall-clock time between the client starting

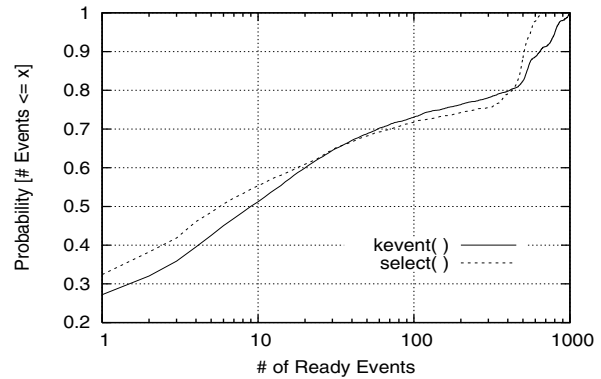


Figure 1: CDF of number of ready events (the return values from `select()`) in Flash

the HTTP request and receiving the last byte of the response. We normally report mean response time, but we note that it can hide the details of the latency profiles, especially under workloads with widely-varying request sizes. So, in addition to mean response time, we also present the 5th, 50th (median) and 95th percentiles of the latency distribution. Where appropriate, we also provide the cumulative distribution function (CDF) of the client-perceived latencies.

3 Blocking in Web Servers

In this section we investigate the origins of the high latency we saw on the earlier tests. By instrumenting the kernel, we trace much of the root cause to blocking in filesystem-related system calls. This blocking affects the queuing model for the services, causing a policy degradation when head-of-line blocking occurs. We present evidence that this behavior is occurring in both Flash and Apache, although via different mechanisms.

3.1 Observing Blocking in Flash

Using our workloads, we find that the main Flash process is blocking inside the kernel on operations other than the `select()` or `kevent()` and the system shows idle CPU time. While CPU idle time is not surprising for a workload that accesses disk, the main process in Flash should never block – all the disk activity should be channeled to the helpers.

Examining the number of ready file descriptors returned per invocation of `select()` or `kevent()` provides more evidence of blocking. These calls form the main loop of an event-driven server, and are invoked as many times as needed as long as the system is active. Event handlers take corresponding actions based on the returned file descriptor value and action indicator. The number of ready descriptors returned by the `select()`

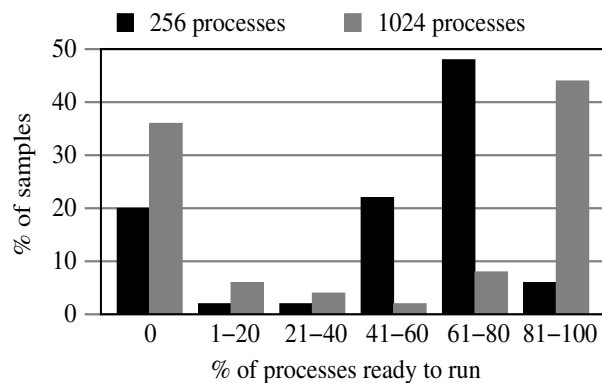


Figure 2: Scheduler burstiness (via the instantaneous run queue lengths) in Apache for 256 and 1024 server processes

or `kevent()` call reflects the queue length which will be processed by event handlers. The CDF of the number of ready descriptors is shown in Figure 1, and indicates that these calls typically return a large number of ready events per call. For `select()`, the median number of ready descriptors is 12, the mean is 61 and the maximum length is more than 600. More than 25% of the invocations return over 100 ready descriptors. The distribution for `kevent()` is similar.

In this workload, the CPU should never be idle – even if the amount of work available decreases, the main loop should call `select()` or `kevent()` more often, decreasing the number of ready descriptors per call. Only when one ready descriptor is returned per call should the CPU exhibit any idle time. However, given the idle time and the observed blocking, we can see that the blocking is causing both the CPU idle time and the batching. Even though descriptors are ready for servicing and idle CPU exists, the blocking system calls are artificially limiting performance and increasing latency.

3.2 Inferring Blocking in Apache

Directly observing a similar problem in Apache is more difficult because any of its processes may block on disk activity, and its multiple-process design exploits the fact that the OS will schedule another process when the running process blocks. While conventional wisdom holds that such blocking is necessary and affects only the request being handled, excess blocking may hinder parallelism and cause high latency.

Since Apache does not have easily-testable invariants regarding blocking such as Flash does, we use another mechanism to infer it. We can use the observation that blocking in Flash increases the burstiness of system activity to find a similar behavior in Apache. In particular, we note that if resource contention occurs in Apache,

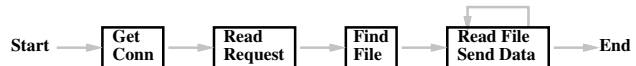


Figure 3: HTTP request processing steps

it would block other processes requesting the same resource, and the release of a resource would involve several processes becoming runnable at the same time. We expect that as more processes are involved, burstiness increases as does run queue variability.

We instrument the OS scheduler to report the number of runnable Apache processes, and test in two configurations. We use 256 and 1024 maximum server processes, an infinite-demand workload, and 1024 clients. Both configurations show roughly the same throughput, due to the infinite-demand model and LAN clients. In Figure 2, we show what percentage of the Apache processes are runnable at any given time.

In both cases, the distribution is very bimodal – most of the time, either no Apache processes are runnable or most of them are. The burstiness, when many processes suddenly become runnable at once, is more evident in the 1024 process case – all processes are blocked roughly one-third of the time, and over 80% of the processes are in the runnable queue over 40% of the time. The 256 process case is only slightly less bursty, with the run queue generally containing 60-80% of the total processes. Note that all processes being blocked does not imply the entire system is idle – disk and interrupt-driven network activity is still being performed in the kernel’s “bottom-half.”

3.3 Causes of Blocking

Our earlier work on developing the DeBox tool [17] identified the call sites in Flash where blocking occurred, but did not investigate the mechanisms by which it occurred. Among the problems, we identified that the Flash server would sometimes block in the “find file” step of the HTTP processing pipeline shown in Figure 3. This step involves performing a series of `open()` and `stat()` calls to traverse the URL’s components in the filesystem. This blocking was unexpected because of the way Flash opens files – it invokes a helper process to perform the steps first, and then the helper notifies the main process, which repeats the process. In this case, the helper had presumably just finished this process, so all of the necessary metadata should have been memory-resident when the main process performed the same actions. This blocking occurs even if the filesystem is mounted asynchronous or read-only, ruling out synchronous metadata writes.

Further investigation reveals that the metadata locking problem is due to lock contention during disk access. In particular, we find that one of the problems is lock con-

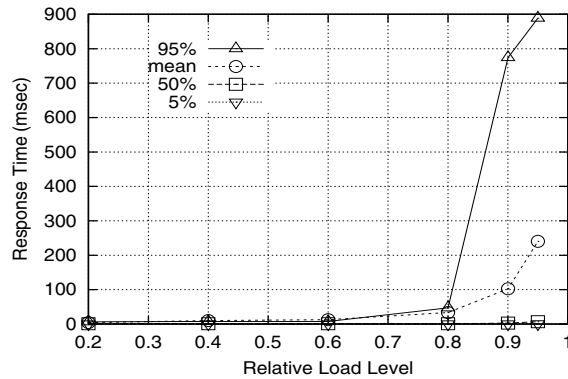


Figure 4: Apache Latency Profile. The relative load of 1.0 equals 241 Mb/s

tention when the main process and the helper access a shared file path. When this happens, the helper usually is doing disk I/O but still holding the vnode name lock to ensure the consistency of the corresponding entry. The decision to make this lock exclusive instead of read-only appears to be a design decision to simplify the associated code – in most types of code, the probability of lock contention would be low, so making this lock exclusive simplifies the code. We further validate this theory by confirming that the blocking occurs even when access time modifications are disabled and even when the filesystem is mounted read-only.

The problem of metadata handling is not FreeBSD-specific. We observe little lock contention in Linux but have observed metadata cache misses commonly occurring when the data set exceeds the physical memory size, causing blocking in otherwise cached requests.

The degree of this problem is significant in FreeBSD due to an interaction between a number of implementation choices. The choices and possible motivation for each are as follows:

- **Exclusive vnode locks** – FreeBSD often uses exclusive vnode locks, presumably to reduce complexity and also to avoid possible deadlock scenarios related to lock promotion.
- **Directory walk locks** – When walking a directory to find a file, the OS acquires the child directory's lock before releasing the parent's lock.
- **Locks during disk access** – If getting the child's inode requires a disk access, the parent's lock is held during the disk access. Releasing locks and re-acquiring them after the disk access causes extra work, and is subject to problems if a higher-level path component changes during the disk access.

These choice are independently reasonable, but their combination leads to the unintended blocking. In partic-

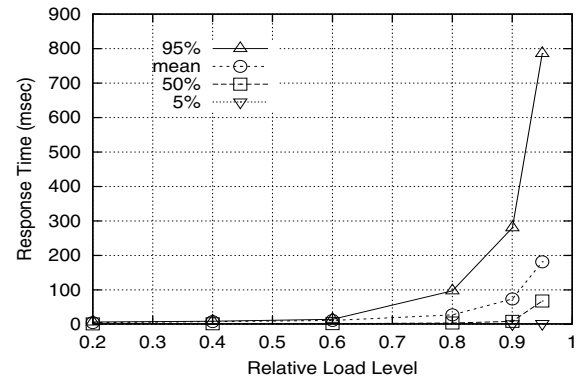


Figure 5: Flash Latency Profile. The relative load of 1.0 equals 336 Mb/s

ular, if multiple processes are trying to resolve similar paths, and one blocks on an inode access, the others can block waiting on an exclusive lock for the shared parent. If more processes try to resolve the same path, they can block higher in the file tree waiting on other readers to release lower-level exclusive locks. A single inode read can then cause many readers to become unblocked, leading to a burst of activity in the form of ready processes.

The metadata locking problem also explains what occurs in Apache and why it has gone unnoticed for so long. Since Apache does not cache open file descriptors, every request processed must perform this same set of steps. The design relies on the OS's own metadata caching to avoid these steps requiring excessive disk access, but without any information about which accesses should be cached, Apache developers can not determine when blocking during an `open()` call is unexpected.

3.4 Response Time Effects

To measure server latency characteristics on disk-bound workloads and show the impact of the underlying blocking problems, we run the servers with request rates of 20%, 40%, 60%, 80%, 90%, and 95% of their respective infinite-demand rates. The results, shown in Figures 4 and 5, show some interesting trends. While the general shape of the mean response curves is not surprising, some important differences emerge when examining the others. Apache's median latency curve is much flatter, but rises slightly at the 0.95 load level (95% of the infinite-demand rate). The mean latency for Apache becomes noticeably worse at that level, with a value comparable to that of Flash, while Apache's latency for the 95th percentile grows sharply.

Some insight into the latency degradation for these servers can be gained by examining the spread of request latencies at the various load levels, shown in Figures 6 and 7. Both servers exhibit latency degradation as

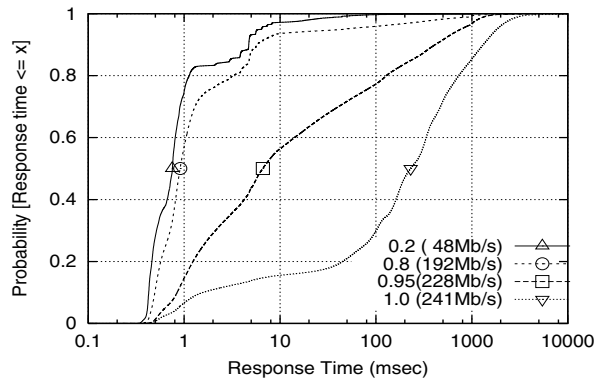


Figure 6: Apache latency CDFs for various load levels

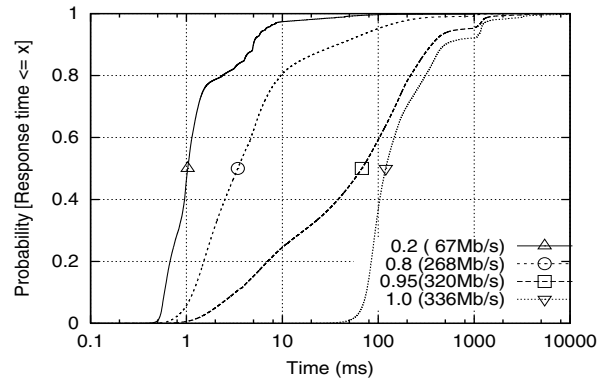


Figure 7: Flash latency CDF for various load levels

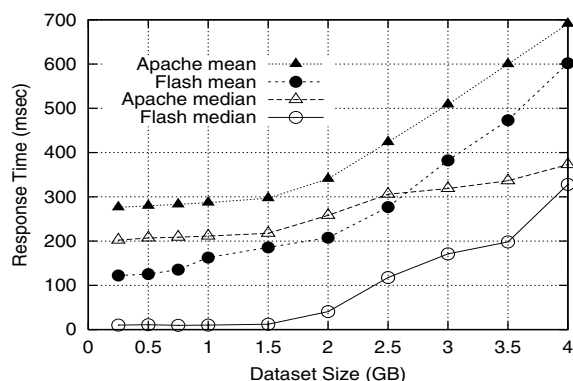


Figure 8: Median and mean latencies of Apache and Flash with various data set sizes

the server load approaches infinite demand, with the median value rising over one hundred times. Two features which appear to be related to the server architecture and blocking effects are immediately apparent – the relative smoothness of the Flash curves, and the seemingly lower degradation for Apache at or below load levels of 0.95. By multiplexing all client connections through a single process, the Flash server introduces some batching effects, particularly when blocking occurs. This batching causes even the fastest responses to be delayed. As a result, Flash returns very few responses in less than 10ms when the load exceeds 95%, whereas Apache still delivers over 60% of its responses within that time. We believe that under low lock contention, Apache’s multiple processes allow in-memory requests to be serviced very quickly without interference from other requests. At higher loads, locking becomes more significant, and only 18% of requests can be served within 10ms.

However, this portion of the CDF does not explain Apache’s worse mean response times, for which the explanation can be seen in the tail of the CDFs. Though Apache is generally better in producing quick responses under load, latencies beyond the 95th percentile grow

sharply, and these values are responsible for Apache’s worse mean response times. Given the slow speed of disk access, these tails seem to be disk-related rather than purely queuing effects. Given the high cost of disk access versus memory speeds, these tails dominate the mean response time calculations.

3.5 Response Time vs. Data Set Size

A deeper investigation of the effect of data set size on server latency provides more insight into the blocking problems as well as a surprising result. Figures 8 shows mean and median latencies as functions of data set size. The mean latency remains relatively flat for the in-memory workload, but begins to grow when the data set size exceeds the physical memory of the machine, 1GB. This increase in mean latency is expected, since these filesystem cache misses require disk access, and the disk latency will raise the mean.

The increase in median latency is quite surprising for this workload – the measured cache hit rate is more than 99%, suggesting that most requests should be comfortably served out of the filesystem cache. The cache hit rate is in line with what we showed in Table 2. These tests confirm that the small amount of cache miss activity is interfering with accesses that should be cache hits.

This observation is problematic, because it implies that, for non-trivial workloads, server latency is tied to disk performance, even for cached requests. Without server or operating system modification, latency scalability is therefore tied to mechanical improvements, rather than faster improvements in electronic components. The expected latency behavior would have been precisely the opposite – that as the number of disk accesses increased, and the overall throughput decreased, the median latency would actually decrease since fewer requests would be contending for the CPU at any time. Queuing delays related to CPU scheduling would be mitigated, as would any network contention effects.

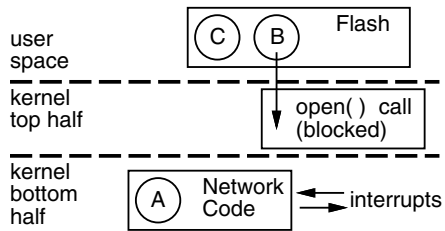


Figure 9: Service inversion example – Assume three requests (A, B, and C) arrive at the same time, and A is processed first. If it is cached and is sent to the networking code in the kernel bottom half, interrupt-based processing for it can continue even if the the process gets blocked. In this case, even if A is large, it may get finished before processing on C even starts.

4 Service Inversion

The most significant effect of this blocking behavior is unnecessary delays in serving queued requests. In particular, cached requests that could have been served in memory and with low latency are forced to wait on disk-bound requests, similar to the priority inversion problem in scheduling. We term this phenomenon “service inversion” since the resulting latencies would be inverted compared to the ideal latencies. In this section, we study this phenomenon and propose an approach to quantify the service inversion value.

Since certain request processing steps operate independently of the server process, any blocking that occurs early in request processing can affect the system’s fairness policies. Specifically, the networking code is split in the kernel, with the sockets-related operations occurring in the “top half”, which is invoked by the application. The “bottom half” code is driven by interrupts, and performs the actual sending of data. So, when an application is blocked, any data that has already been sent to the networking code can still operate in the kernel’s “bottom half.” Likewise, since the disk helpers in Flash operate as separate processes, they can continue to operate on their current request even when the main process is blocked.

Head-of-line blocking in the literature is usually studied in the network scheduling context. To understand the blocking scenario in the OS and how it causes service inversion, consider the scenario in Figure 9, where three requests arrive simultaneously, with the middle request causing the process to block. Assume it is blocked by an `open()` call, which takes place before the data reads occurs (if needed) and before any data is sent to the networking code. If the first and third requests are cached, they would normally be served at nearly the same time. However, the first request may get sent to the networking code, and the third request would then have to wait until the process is unblocked. The net effect is that the third

series	size range	percentage
1	0.1 - 0.5 KB	25.06%
2	0.6 - 4 KB	28.05%
3	5 - 6 KB	23.55%
4	7 - 900KB	23.34%

Table 3: Workload categories for latency breakdowns. Percentages shown are the dynamic request frequencies for the given file sizes in the SPECWeb99 workload.

request suffers from head-of-line blocking. The system’s fairness policies, particularly the scheduling of network packets, are not given a chance to operate since the three requests do not reach the networking code at the same time.

If the requests before the blocked requests are larger than the ones that follow, we label the resulting phenomenon *service inversion*. The occurrence of this behavior is relatively simple to detect at the client – the latencies for small requests would be higher than the latencies for larger requests.

4.1 Identifying Service Inversion

To qualitatively understand the prevalence of service inversion, we take the latency CDFs from Figures 6 and 7 and split them by decile. Since SPECWeb biases toward small files and more than 95% of the requests could fit into physical memory, ideal response times would be roughly proportional to transfer sizes. By examining the different response sizes within each decile, we can estimate the extent of reordering. To simplify the visualization, we group the responses by sizes into four series such that their dynamic frequencies are roughly equal. The details of this categorization are shown in Table 3.

The graphs in Figures 10 and 11 show the composition of responses by decile for the two servers, with the leftmost bar corresponding to the fastest 10% of the responses and the rightmost representing the slowest 10%. These graphs are taken from the latency CDFs at a load level of 0.95.

In a perfect scenario with no service inversion, the first 2.5 bars would consist solely of responses in Series 1, followed by 2.5 bars from Series 2, etc. However, both graphs show responses from the different series spread across all deciles, suggesting both servers exhibit service inversion. One surprising aspect of these plots is that the Series 1 values are spread fairly evenly across all deciles, indicating that even the smallest files are often taking as long as some of the largest files.

Some inversion is to be expected from the characteristics of the workload itself, since directories are weighted according to a Zipf-1 distribution. With roughly 600 directories in our data set, the last directory receives 600

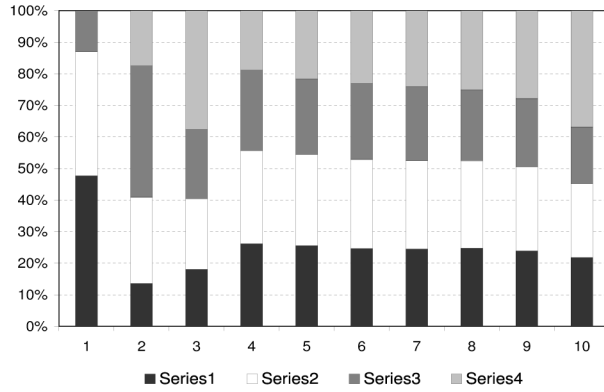


Figure 10: Apache CDF breakdown by decile at load level 0.95

times fewer requests than the first. So, even though files 100KB or greater account for only 1% of the requests (35 times fewer than the smallest files), the directory bias causes the largest files in the first directory to be requested about 17 times as frequently as the smallest files in the final directory. While the large files still require much more space, an LRU-style replacement in the filesystem cache could cause these large files to be in memory more often. In practice, this effect is relatively minor, as we will show later in the paper.

4.2 Quantifying Service Inversion

While the latency breakdowns by decile qualitatively show the system's unfairness, a more quantitative evaluation of service inversion can be derived from the CDF. We construct the formula based on the following observation: Given responses A, B, C, D, E with sizes $A < B < C < D < E$. If the observed response times have the same order as the response sizes, we say that no service inversion has occurred, and the corresponding value should be zero. On the contrary, if the response times are in the reverse order of their sizes, then we say that the server is completely inverted, and give it a value of 1.

The insight into calculating the inversion is as follows: we want to determine how perturbed a measured order is, compared with the order of the response sizes. Perturbation is the difference in position of a response in the ordered list of response times versus its position in a list ordered by size, where the per-response distances are summed for the entire list. We then normalize this versus the maximum perturbation possible. A particular service inversion value is given by:

$$\sum_{i=1}^n \text{Distance}(i) / \lfloor n^2/2 \rfloor \quad (1)$$

where distance is absolute value of how far the request is from the ideal scenario, and $\lfloor n^2/2 \rfloor$ is the to-

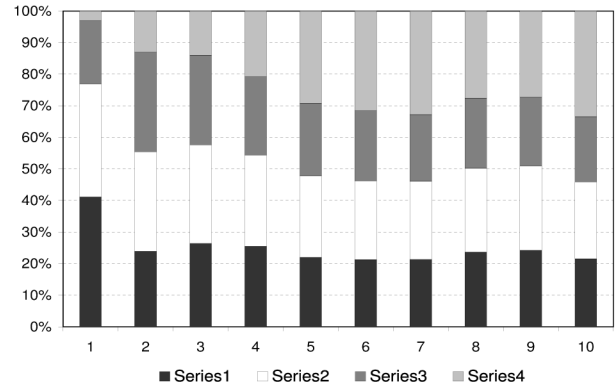


Figure 11: Flash CDF breakdown by decile at load level 0.95

	Relative load level					
	0.20	0.40	0.60	0.80	0.90	0.95
Apache	0.14	0.23	0.28	0.51	0.54	0.58
Flash	0.25	0.35	0.45	0.52	0.56	0.58

Table 4: Service inversion versus load level

tal distance of requests in the reverse order of their sizes, which is the maximum perturbation possible. In the above example, assume the observed latency order is B, C, A, D, E . By comparing with the ideal order, A, B, C, D, E , we see the distance of file B is 1, C is 1, A is 2, and D, E are 0. The inversion value is $4/12 = 0.33$. Since this measurement requires only the response sizes and latencies, as long as the distribution of sizes is the same, it can be used to compare two different servers or the same server at multiple load levels. To handle the case of multiple requests with the same response size, we calculate distance by comparing the N^{th} observed position with the N^{th} ideal position for each response of the same size.

By measuring service inversion as a function of load level, we discover that this effect is a major contributor to the latency increase under load. Table 4 shows the quantified inversion values for both servers, and demonstrates that while inversion is relatively small at low loads, it exceeds half of the worst-case value as the load level increases. The latencies at the higher load levels therefore not only suffer from queuing delays, but also service inversion delays from blocking. We will show in the next section that the delays stemming from blocking and service inversion are in fact the dominant source of delay.

5 The New Servers & Results

In this section we describe our solution and evaluate the resulting systems. We analyze the effects on capacity, latency, and service inversion, and demonstrate that our

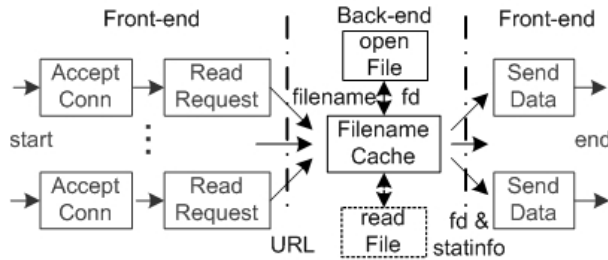


Figure 12: Flashpache architecture

new servers overcome the latency and blocking problems previously observed. In our earlier work on DeBox [17], we modified the Flash Web server to avoid blocking. We briefly describe those changes to provide the context for our new results with Apache.

Since the blocking has multiple origins, we believe a portable user-level process is preferable to invasive kernel changes. Accordingly, we modify both servers to reduce blocking. Our new contribution in this respect is to identify how Apache can be easily modified to take advantage of the same kinds of changes that helped Flash. Additionally, we focus on latency and service quality evaluation of the resulting servers, in order to understand how the new techniques work.

Our earlier changes to Flash focus on detecting and avoiding blocking, or moving blocking out of the main server process. The `open()` and `stat()` calls are moved entirely out of the main process, and the helpers return file descriptors and metadata information to the main process using the `sendmsg()` system call. Thus, the main process can operate continuously without blocking. Data copying is eliminated by using `sendfile()` instead of `writew()`, and the memory-mapping calls that were used in conjunction with `writew()` are eliminated. Finally, some other changes are made to `sendfile()` to reduce its memory usage and eliminate disk access. We term the resulting server New-Flash.

5.1 Flashpache

Due to the differences in software architecture, we cannot directly employ the same techniques that we used in New-Flash to improve Apache. However, given our earlier measurements on Apache, we can deduce that filesystem-related calls are likely to block, and with these as candidates, we can leverage the lessons from Flash. Since Apache does not cache file descriptors, each process calls `open()` on every request, and this behavior results in a much higher rate of these calls.

We modify Apache to offload the URL-to-file translation step, in which metadata-related system calls occur. This step is handled by a new “backend” process, to

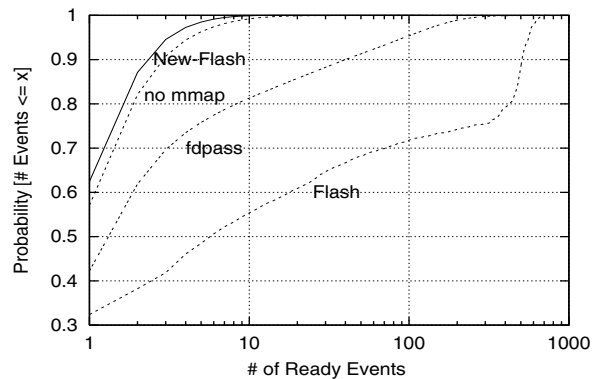


Figure 13: CDFs of # of ready events for Flash variants, infinite-demand workload

which all of the Apache processes connect via persistent Unix-domain sockets. The backend employs a Flash-like architecture, with a main process and a small number of helpers. The main process keeps a filename cache like the one in the Flash server, and schedules helpers to perform cache miss operations. The backend is responsible for finding the requested file, opening the file, and sending the file descriptor and metadata information back to the Apache processes. Upon receiving a valid open file descriptor from the backend, the Apache process can return the associated data to the client. Since the backend handles URL lookup for all Apache processes, it is possible to combine duplicated requests and even preload data blocks into the filesystem cache before passing control back to the Apache processes, thus reducing the number of context switches and the chances of blocking.

We call this new server Flashpache, to reflect its hybrid architecture. The changes involved in this process are relatively small and isolated – fewer than 100 lines of code are modified in Apache, and half of this count is code taken directly from New-Flash. The backend process is similarly derived from parts of New-Flash, and consists of roughly 200 lines of code changes.

This architecture, shown in Figure 12, eliminates unnecessary blocking in two ways. First, in Flashpache, most of the disk access is performed by a small number of helper processes controlled by the backend, reducing the amount of locking contention. This observation is confirmed by the fact that less blocking occurs in Flashpache than in Apache with the same workload. Second, since the backend caches metadata information and keeps files open, it effectively prevents metadata cache entries from being evicted when memory pressure is an issue. However, we do not observe the CPU reduction from caching as the main source of the benefit – the interprocess communication cost between the Apache processes and the backend is almost equivalent to or even a little higher than the original system calls.

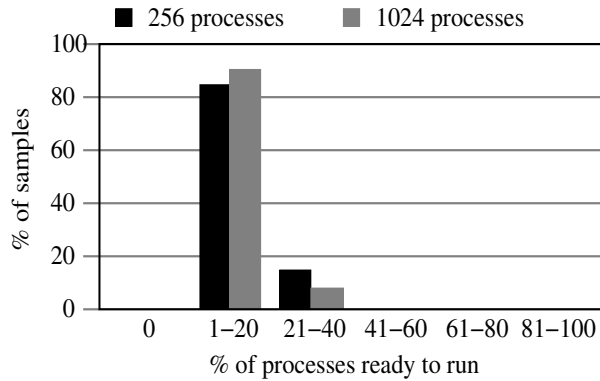


Figure 14: Scheduler burstiness in Flashpache for 256 and 1024 processes, infinite-demand workload

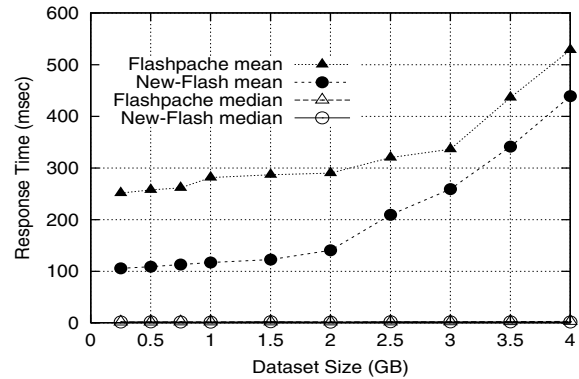


Figure 15: Response times for new servers with different data set sizes and infinite-demand workload

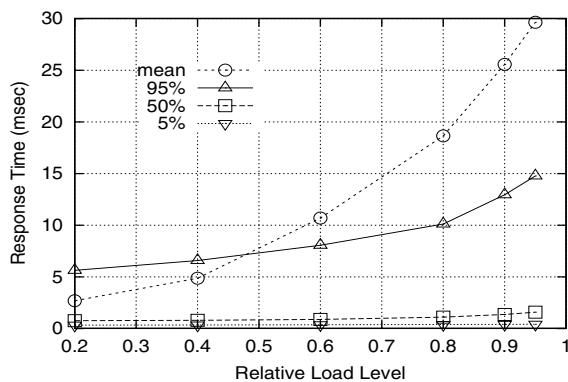


Figure 16: Latency profile of New-Flash (Flash profile shown in Figure 5). Load level 1.0 equals 450 Mb/s

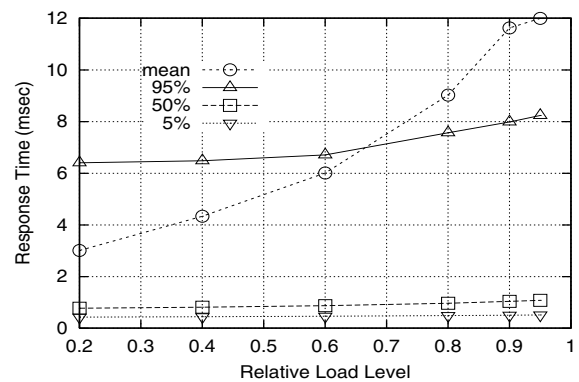


Figure 17: Latency profile of Flashpache (Apache profile shown in Figure 4). Load level of 1.0 equals 273 Mb/s

	Latency (ms)			Capacity (Mb/s)
	median	mean	90%	
Flash	67.4	181.0	362.0	336.0
fd pass	11.5	50.0	71.2	395.0
no mmap	1.8	93.5	92.9	437.5
New-Flash	1.6	29.3	6.6	450.0
Apache	6.6	180.2	414.7	241.1
Flashpache	1.1	12.0	5.7	272.9

Table 5: Latencies & capacities for all servers

5.2 Latency Results

We analyze the latency of the new servers by repeating our earlier experiments to understand latency and blocking. We begin by repeating the burstiness measurement, which indicates that blocking-induced burstiness has also been reduced or eliminated in both servers. In Figures 13 and 14, we see that in New-Flash, the mean number of events per call has dropped from 61 to 1.6, and the median has dropped from 12 to 2. Likewise, Flashpache no longer exhibits bimodal behavior at the scheduler level, instead showing roughly 20% of all processes ready at

any given time. In both cases, the request batching and associated idle periods are eliminated.

We evaluate step-by-step improvements to Flash with the results shown in Table 5. Included are the figures for the original Flash, as well as the intermediate steps of file descriptor passing (fd pass) and removing memory-mapped files (no mmap). Throughputs are measured with infinite-demand and response times are measured at 0.95 load level. We can see that the overall capacity of Flash has increased by 34% for this workload, while Apache's capacity increases by 13%.

The more impressive result is the reduction in latency, even when run at these higher throughputs. Flash sees improvements of 40x median, 6x mean, and 54x in 90th percentile latency. Eliminating metadata-induced blocking has improvements of 5.8x median, and 3.6x mean, and eliminating blocking in `sendfile()` reduces a factor of 3 in mean latency. Apache sees improvements of 6x median, 15x mean, and 72x in 90th percentile latency. The one seemingly odd result, an increase in mean latency from fd-pass to no-mmap, is due to an increase in blocking, since the removal of `mmap()` also results

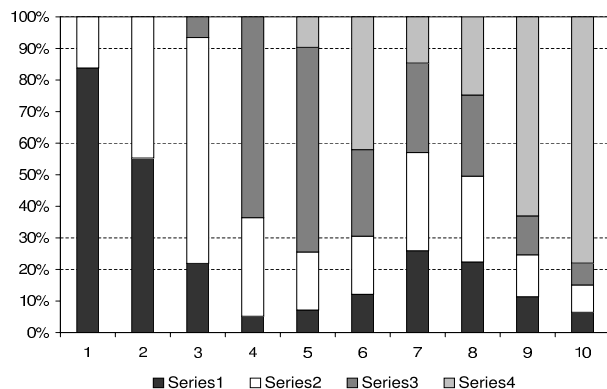


Figure 18: CDF breakdown for New-Flash on 3.0 GB data set, load level 0.95

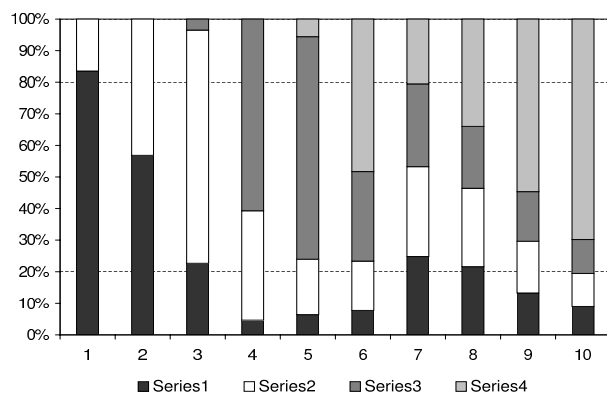


Figure 20: CDF breakdown for New-Flash on in-memory workload, load level 0.95

in losing the `mincore()` function, which could precisely determine memory residency of pages. The New-Flash server obtains this residency information via a flag in `sendfile()`, which again eliminates blocking.

Not only do the new servers have lower latencies, but they also show qualitatively different latency characteristics. Figure 15 shows that median latency no longer grows with data set size, despite the increase in mean latencies. Mean latency still increases due to cache misses, but the median request is a cache hit in all cases. Figures 16 and 17 show the latency CDFs for 5th percentile, mean, median, and 95th percentile with varying load. Though the mean latency and 95th percentile increase, the 95th percentile shows less than a tripling versus its minimum values, which is much less than the two orders of magnitude observed originally. The other values are very flat, indicating that most of the requests are served with the same quality at different load levels. More importantly, the 95th percentile CDF values are lower than the mean latency, because the time spent on the largest requests (the last 5%) is much higher than the time spent on other requests, as expected from Table 2.

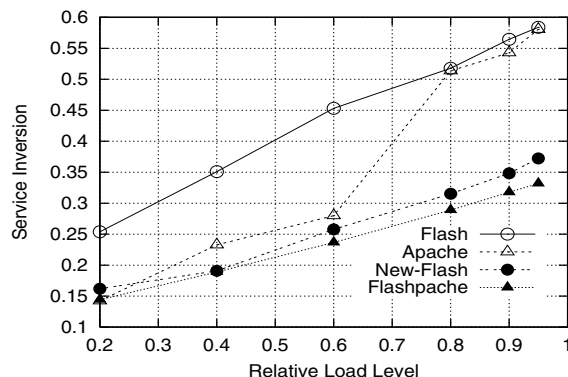


Figure 19: Service inversion of original and modified servers

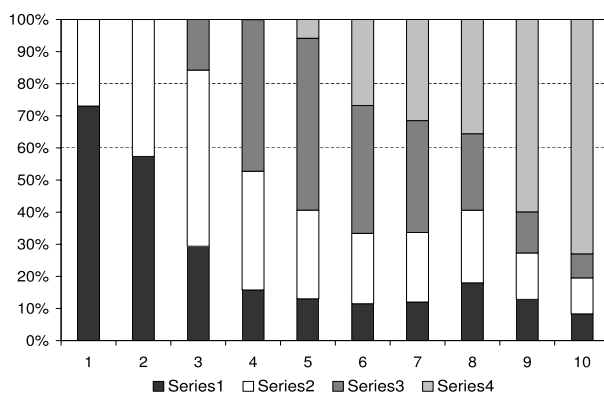


Figure 21: CDF breakdown for Flashpache on 3.0 GB data set, load level 0.95

5.3 Service Inversion Improvements

In order to verify the unfairness of the new servers, we further examine the latency breakdown by decile for the 0.95 relative load level and the service inversion at different load levels. Figure 18 shows the percentage of each file series in each decile for New-Flash, and we observe some interesting changes compared to the original server. The smallest files (series 1) dominate the first two deciles, the largest files (series 4) dominate the last two deciles, and the series 3 responses are clustered around the fifth decile. This behavior is much closer to the ideal than what we saw earlier. Some small responses still appear in the last column, but these may stem from files with low popularity incurring cache misses. Also complicating matters is that the absolute latency value is now below 10ms for 98% of the requests, so the first nine deciles are very compressed. This observation is verified by calculating the service inversion value.

Figure 19 shows the change of the inversion value with the load level. Compared to the old system, we reduce the inversion by over 40%, suggesting requests are treated more fairly in the new system. The fact that the

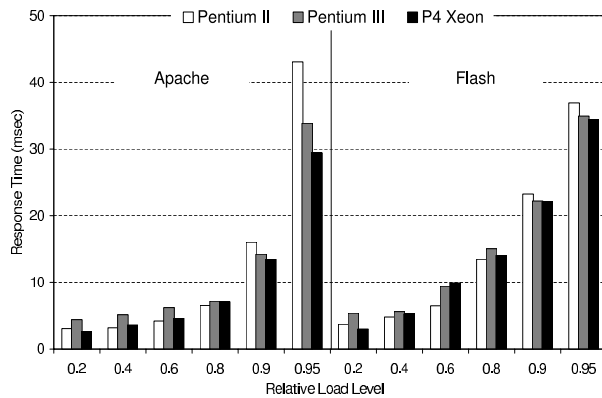


Figure 22: In-memory workload (0.5 GB) latencies of Apache and Flash across three processor generations

	Pentium II	Pentium III	Pentium 4
In-memory workload (0.5GB) capacity in Mb/s			
Apache	107.3	248.4	437.6
Flash	210.3	466.0	787.0
Disk-bound workload (3.0GB) capacity in Mb/s			
Apache	98.8	174.1	241.1
Flash	134.1	256.4	336.0
Flashpache	103.3	198.9	272.9
New-Flash	140.4	358.0	450.0

Table 6: Capacities of original and modified servers across three processor generations and two workloads

inversion value still increases with the load is a matter for further investigation. However, this may be a limitation of our service inversion calculation itself.

By comparing service inversion for this workload with that of a completely in-memory workload, we can see how far we are from a nearly “ideal” scenario. In particular, we are still concerned whether filesystem cache misses are responsible for the service inversion. Figure 20 shows the latency breakdown for a workload with a 500MB data set. The difference between it and the New-Flash breakdown are visible only after careful examination. The numerical value for the in-memory case is 0.33, while the New-Flash result is 0.35, suggesting that if any inversion is due to cache misses, its measured effects are minimal. The Flashpache breakdown, shown in Figure 21, is similar. The values for Flashpache and its original counterpart are also shown in Figure 19, and we can see that our modifications have almost halved the inversion under high load.

5.4 Latency Scalability

To understand how latencies are affected by processor speed, we use three generations of hardware with various processor speed but sharing most of the other hard-

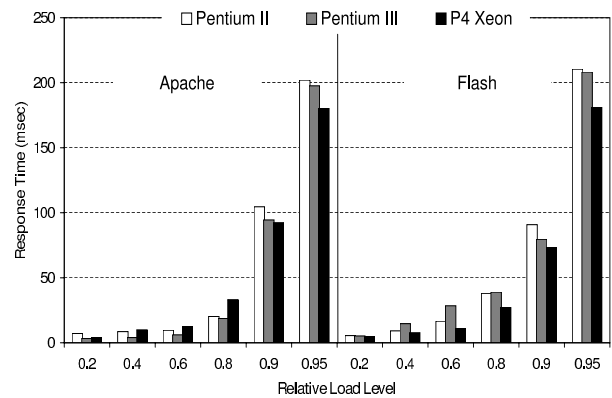


Figure 23: Disk-bound workload (3.0 GB) latencies of Apache and Flash across three processor generations

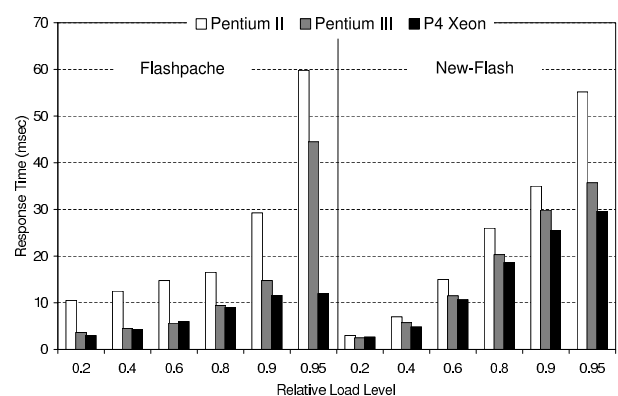


Figure 24: Disk-bound workload (3.0 GB) latencies of New-Flash and Flashpache across three generations

ware components. Details about our server machines are shown in Section 2. We begin our study by measuring the infinite-demand capacity of the two original servers while adjusting the data set size. The results, shown in Table 6, indicate that in-memory capacity of both Apache and Flash scales well with processor speed. But once the data set size exceeds physical memory, performance degrades. Even though the heavy-tailed 3GB Web workload only requires reasonable amount of disk activity, we observe the two faster processors have idle CPU, suggesting performance is tied to disk performance on this workload.

A more detailed examination of server latency is shown in Figures 22 and 23. These two graphs represent an in-memory workload and a disk-bound workload, respectively, and show the mean latencies for both server packages across all three processors. Measurements are taken at various load levels, and show a remarkable consistency – at the same relative load levels, both Apache and Flash exhibit similar latencies, the in-memory latencies are much lower than the disk-bound latencies, and the latencies show only minor improvement with pro-

cessor speed. Figure 24 shows the scalability of our new servers across processors – even with much lower Pentium-II latencies, improvements in processor speed now reduce latency on both servers. This result confirms that once blocking is avoided, the servers can take more advantage of improvements in hardware performance.

In summary, both new servers demonstrate lower initial latencies, slower latency growth, and better decrease of latency with processor speed. These servers are no longer dominated by disk access times, and should scale with improvements in processors, memory, etc. The fact that these changes eliminate over 80% of the latency answers the question about latency origins – these latencies were dominated by blocking, rather than request queuing.

6 Related Work

Performance optimization of network servers has been an important research area, with much work focused on improving throughput. Some addressed coarse-grained blocking – e.g. Flash [13] demonstrated how to avoid some disk-related blocking using non-blocking system calls. Much evaluation about disk I/O associated overheads has focused on Web proxies [10]. Some of the most aggressive designs have used raw disk, eliminated standard interfaces, and eliminated reliable metadata in order to gain performance [18]. In comparison, we have shown that no kernel or filesystem changes are necessary to achieve much better latency, and that these techniques can be retrofitted to legacy servers with low cost.

More recently, much attention is paid to latency measurement and improvement. Rajamony & Elnozahy [15] measure the client-perceived response time by instrumenting the documents being measured. Bent and Voelker explore similar measurements, but focus on how optimization techniques affect download times [3]. Improvement techniques have been largely limited to connection scheduling, with most of the attention focused on the SRPT policy [4, 5]. Our work examines the root cause of the blocking, and our solutions subsume any need for application-level connection scheduling. Our new servers use the existing scheduling within the operating system, and the results suggest that eliminating the obstacles yields automatic improvement with existing service and fairness policies.

Synchronization-related locking has been a major concern in parallel programming research. Rajwar et al. [16] proposed a transactional lock-free support for multi-threaded systems. The reasons of locking in our study have a broader range and differ in application domain. While head-of-line blocking is a well-known phenomenon in the network scheduling context, e.g. Puente et al. [14] and Jurczyk et al. [8] studied various blocking

issues in network environment, we demonstrate that this phenomenon also exists in network server applications and has severe effects on user-perceived latency.

This paper also takes a different approach to fairness than other work, and the difference may be important in some contexts. The SEDA approach [21] tried to schedule requests based on size, but no measurements are presented on the effectiveness of the scheduling itself. Interestingly, despite the four orders of magnitude variation in SPECWeb's file sizes, SEDA handles 80% of requests in 200-1000ms, with a median of over 500ms, over 10 times slower than Flash or Apache. This uniformly slow response time gives it a high score on the Jain fairness index [7] when fairness is evaluated on a per-client basis. On a per-request level, however, we believe that shorter responses should be served faster, and believe that our service inversion metric is more useful. With coast-to-coast latencies in the continental US on the order of 100ms, and with news sites (Yahoo, CNN, etc.) routinely having over 100 embedded objects per page, SEDA's server-induced latency would be a noticeable problem for real Web use.

At the other extreme, Bansal & Harchol-Balter [2] investigate the unfairness of the SRPT scheduling policy under heavy-tailed workloads and draw the conclusion that the unfairness of their approach is barely noticeable. By addressing the latency issues directly rather than scheduling around them, our approach removes the need for the application to explicitly schedule connections. Network scheduling can still be used, particularly for traffic shaping, prioritization, etc.

Finally, we should mention that the FreeBSD SMPng effort, which is released as FreeBSD 5, has completely rewritten much of the locking in FreeBSD, using finer-granularity locks to improve performance on multiprocessor machines. Additionally, some of the filesystem locking appears to have introduced read-shared locks in addition to exclusive locks. These locks could reduce the chance of lock convoys in pathname resolution, eliminating some of the blocking we observed. While we would like to evaluate the behavior of this system, we have been unable to operate it under sufficient load without kernel panics or significant performance degradation.

7 Conclusion

In this paper, we have examined server latency and traced the root of much of the problem to head-of-line blocking within filesystem-related kernel queues. This behavior may have little impact on throughput, but severely degrades latency and service quality. By examining individual request latencies, we find that this blocking gives rise to a phenomenon we call *service inversion*, where requests are served unfairly.

By addressing the blocking issues both with the Apache and the Flash server, we improve latency by more than an order of magnitude, and demonstrate a qualitatively different change in the latency profiles. We performed these changes in user space, in a portable manner, without requiring any modification to the kernel or filesystem layout. Without much effort or extensive modification, we were able to take advantage of these changes in a widely-deployed legacy server. The resulting servers also exhibit lower burstiness, and more fair request handling. Their latency values scale better with improvements in processor speed than their original counterparts, making them better candidates for future improvements. Finally, our results suggest that most server-induced latency is tied to blocking effects, rather than queuing.

In addition to the practical benefits of this research, the delivery of servers with much better latency properties, this work also improves on our fundamental understanding of the interactions between the filesystem, application, and workloads. By addressing the root causes of latency increase in network servers, we believe that we can enhance research in other areas, such as improving quality of service or scheduler policies.

Acknowledgments

We would like to thank the anonymous reviewers for their useful feedback on the paper. This work was supported in part by NSF grant CNS-0519829.

References

- [1] Apache Software Foundation. The Apache Web server. <http://www.apache.org/>.
- [2] N. Bansal and M. Harchol-Balder. Analysis of SRPT scheduling: Investigating unfairness. In *Proc. of the SIGMETRICS '01 Conference*, Cambridge, MA, June 2001.
- [3] L. Bent, Geoffrey, and M. Voelker. Whole page performance. In *7th International Workshop on Web Content Caching and Distribution (WCW'02)*, Boulder, CO, Aug. 2002.
- [4] M. Crovella, R. Frangioso, and M. Harchol-Balder. Connection scheduling in web servers. In *Proc. of the 2nd USENIX Symp. on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.
- [5] M. Harchol-Balder, B. Schroeder, M. Agrawal, and N. Bansal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems*, 21(2), May 2003.
- [6] J. C. Hu, I. Pyrali, and D. C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *Proceedings of the 2nd Global Internet Conference*, Phoenix, AZ, Nov. 1997.
- [7] R. Jain. Congestion control and traffic management in ATM networks: Recent advances and A survey. *Computer Networks and ISDN Systems*, 28(13):1723–1738, 1996.
- [8] M. Jurczyk and T. Schwederski. Phenomenon of higher order head-of-line blocking in multistage interconnection networks under nonuniform traffic patterns. *IEICE Transactions on Information and Systems*, E79-D(8):1124–1129, August 1996.
- [9] J. Larus and M. Parkes. Using cohort-scheduling to enhance server performance. In *USENIX 2002 Annual Technical Conference*, pages 103–114, Monterey, CA, June 2002.
- [10] E. P. Markatos, M. Katevenis, D. N. Pnevmatikatos, and M. Flouris. Secondary storage management for web proxies. In *Proc. of the 2nd USENIX Symp. on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.
- [11] L. W. McVoy and C. Staelin. Imbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference*, pages 279–294, San Diego, CA, June 1996.
- [12] Mindcraft, Inc. WebStone Benchmark. <http://www.mindcraft.com/webstone>.
- [13] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX 1999 Annual Technical Conference*, pages 199–212, Monterey, CA, June 1999.
- [14] V. Puente, J. A. Gregorio, C. Izu, and R. Beivide. Impact of the head-of-line blocking on parallel computer networks: Hardware to applications. In *European Conference on Parallel Processing*, pages 1222–1230, 1999.
- [15] R. Rajamony and M. Elnozahy. Measuring client-perceived response times on the www. In *Proc. of the 3rd USENIX Symp. on Internet Technologies and Systems*, San Francisco, CA, March 2001.
- [16] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2002.
- [17] Y. Ruan and V. Pai. Making the “box” transparent: System call performance as a first-class result. In *Proc. of the USENIX 2004 Annual Technical Conference*, Boston, MA, June 2004.
- [18] E. Shriver, E. Gabber, L. Huang, and C. A. Stein. Storage management for web proxies. In *Proc. of the USENIX 2001 Annual Technical Conference*, pages 203–216, Boston, MA, 2001.
- [19] Standard Performance Evaluation Corporation. SPEC Web 96 & 99 Benchmarks. <http://www.spec.org/osg/web96/> and <http://www.spec.org/osg/web99/>.
- [20] R. von Behren, J. Condit, F. Zhou, G. C. Necula, , and E. Brewer. Capriccio: Scalable threads for internet services. In *Proc. of the 18th ACM Symp. on Operating System Principles*, Bolton Landing, NY, Oct. 2003.
- [21] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of the 19th ACM Symp. on Operating System Principles*, pages 230–243, Chateau Lake Louise, Banff, Canada, Oct. 2001.

Reval: A Tool for Real-time Evaluation of DDoS Mitigation Strategies

Rangarajan Vasudevan, Z. Morley Mao
University of Michigan

Oliver Spatscheck, Jacobus van der Merwe
AT&T Labs — Research

Abstract

There is a growing number of DDoS attacks on the Internet, resulting in significant impact on users. Network operators today have little access to scientific means to effectively deal with these attacks in real time. The need of the hour is a tool to accurately assess the impact of attacks and more importantly identify feasible mitigation responses enabling real-time decision making. We designed and implemented *Reval*, a tool that reports DDoS attack impact in real time, scaling to large networks. This is achieved by modeling resource constraints of network elements and incorporating routing information. We demonstrate the usefulness of the tool on two real network topologies using empirical traffic data and examining real attack scenarios. Using data from a tier-1 ISP network (core, access and customer router network) of size in excess of 60000 nodes, *Reval* models network conditions with close to 0.4 million traffic flows in about 11 seconds, and evaluates a given mitigation deployment chosen from a sample set in about 35 seconds. Besides real-time decision support, we show how the simulator can also be used in longer term network planning to identify where and how to upgrade the network to improve network resilience. The tool is applicable for networks of any size and can be used to analyze other network anomalies like flash crowds.

1 Introduction

Denial of Service (DoS) and Distributed Denial of Service (DDoS) attacks are on the increase in today's Internet [1]. They have morphed into extortion tools to attack both small and large businesses alike [2]. Since a large number of customers share the same network infrastructure, attack traffic traversing an ISP network causes service loss not only to the targets of the attacks but also to other customers. Therefore it is important for network operators to mitigate the impact of DDoS attacks

to continue providing guaranteed service. For this, network operators first need to understand the resilience of their networks to attacks, and secondly require a method of determining the best mitigation strategies to follow. Both of these need to be performed in *real time* to be of benefit to the customers. As soon as an attack occurs, attack impact analysis is required to provide real-time network information in terms of which links are overloaded, which customers are affected *etc.* This in turn guides network operators in real-time decision making to evaluate and choose the best combination of mitigation responses from the set of mechanisms available in the network.

To perform attack impact analysis in real time, measurement studies on attacks — either direct [3, 4, 5, 6] or indirect [7] — can be conducted. To obtain a complete network-wide view, it would require instrumenting each router in the network with a measurement setup to collect relevant data resulting in excessive overhead and cost. Even if such measurements are instrumented network-wide, there is still a need to aggregate data to generate the combined report for network operators, requiring prohibitive communication overhead and excessive delay. Furthermore, measurement data alone do not provide information on how to mitigate against attacks as defense options still need to be evaluated and ranked.

Our pragmatic approach is to use simulations combined with partial measurement data collected from the network without requiring complete network link-level measurement instrumentation. Simulations coupled with routing analysis provide a holistic view identifying attack impact on the entire network. To meet real-time requirements, we have realized *informed optimizations* to the common-case tasks such as route computation in our tool, and choose the right granularity to provide attack impact information on the fly.

Deciding the best attack mitigation strategies by trial-and-error through actual network deployment is too costly and time consuming to carry out in real networks. To avoid this expense, we argue again that simulation

is a preferred avenue of pursuit, as it allows low-cost, fast analysis of various attack scenarios and network conditions without actually deploying them on the physical network. In short, our simulator provides both *real-time decision support* as well as *long-term network design analysis* capabilities for operators of even large-scale IP networks involving tens of thousands of network elements with an order of a million traffic flows traversing the network simultaneously.

The rest of the paper is organized as follows. We first describe our tool Reval in Section 2. A few simulation scenarios along with their results for the Abilene network are presented in section 3. Then, our experience using the tool in evaluating defense mechanisms for a large ISP network is presented in Section 4. Finally, we discuss related work in Section 5 and conclude.

2 Reval Description

In estimating the impact of attacks in real time, today's network operator has to consider data of many different types like network topology, routing, traffic matrices *etc.* To obtain the right mitigation for an attack, she has to use her experience and intuition in arriving at an optimal answer from multiple impact estimations. Drawing from our experiences with a large tier-1 ISP, we find that this process is currently manual and very complex to realize in real time. The simple solution of static deployment of defense mechanisms does not work in practice as we demonstrate in Section 4. Therefore, we require a tool that automates the process of impact analysis and evaluates desired mitigation scenarios tailor-made to an attack handling the complexity of the whole task as a routine. Using this tool, the network operator needs only intervene in specifying the mitigation mechanism to evaluate. Then, once the tool finishes execution, the network operator can choose the mitigation deployment scenario that offers the desired defense for her network. We provide Reval as the tool to fulfill this role in any network where mitigation deployment is not immediately obvious.

The goals of an attack analysis tool are to: (i) Evaluate attack impact on a given network; (ii) Evaluate different defense mechanisms; (iii) Emulate real conditions as close as possible (routing algorithms, best practices, network element behavior); (iv) Be configurable, user-friendly, and extensible; (v) Be scalable and efficient, providing *real-time* feedback; (vi) Execute on off-the-shelf workstations with modest resources. No existing tools/simulators achieve all the above goals in a satisfactory manner, as we discuss in Section 5. The overall design of Reval is depicted in Figure 1. In brief, it takes as input the topology of the network under analysis, background traffic (not malicious or bad) information, attack traffic information, attack mitigation policy

configuration, and other preferences for generating various attack impact metrics. The tool provides capabilities for what-if analyses to understand how well suggested mitigation responses work, and also supports long-term network planning.

2.1 Design Space

Before studying the impact of an attack on a network, we first state the type of attacks we consider and how we calculate their impact. We intended Reval to work with any DDoS attacks that cause impact at the network level — either to access links or core links. DoS attacks like SYN floods and ICMP floods exploit protocol behavior and send large traffic volumes destined towards an end-user or a small set of end-users. If these attacks are not large enough to overload any part of the ISP network, then they are not considered in our tool. Having said that, we can use Reval in numerous scenarios including network management scenarios like capacity planning, overprovisioning; other impact scenarios like flash crowds; analyzing target(s)-specific attack impact and defense mitigation. In this paper, we outline one particular use of Reval while its design readily lends itself to any situation where network-level analysis is required.

Given that ISP networks are set up for profits which predominantly comes from customers who pay to use the network, a reasonable way of defining impact is to relate it with the degree to which customers can communicate without noticing any abnormal performance. We chose to capture *this* way of calculating impact in our tool at a per-customer flow level. That is, network impact of an attack is directly related to the number of customer flows impacted due to performance degradation. By computing attack impact at the flow-level, we avoid the need to carry out time-wise expensive packet-level simulations. Also, we omit the notion of a simulation clock in Reval. The main drawback of these modeling decisions is that the tool is unable to provide time-based fine-grained statistics such as percentage of packets dropped, duration of link and router overloads *etc.* However, as we show in Sections 3 and 4, these statistics are not always required for assessing attack impact.

The data flow of Reval is depicted in Figure 1. Note that the input required for simulation is data that is typically collected by network operators in day-to-day operations. Network topology information is directly extracted from router configuration files; traffic information is directly obtained from Netflow data typically generated by routers in the network. Similarly, the availability of commercial DDoS detection systems [8, 9] allow operators to collect attack traffic information. For example, the tier-1 ISP that we studied has a DDoS detection system deployed at key locations, which, together with Netflow and routing data allows us to construct attack flows for

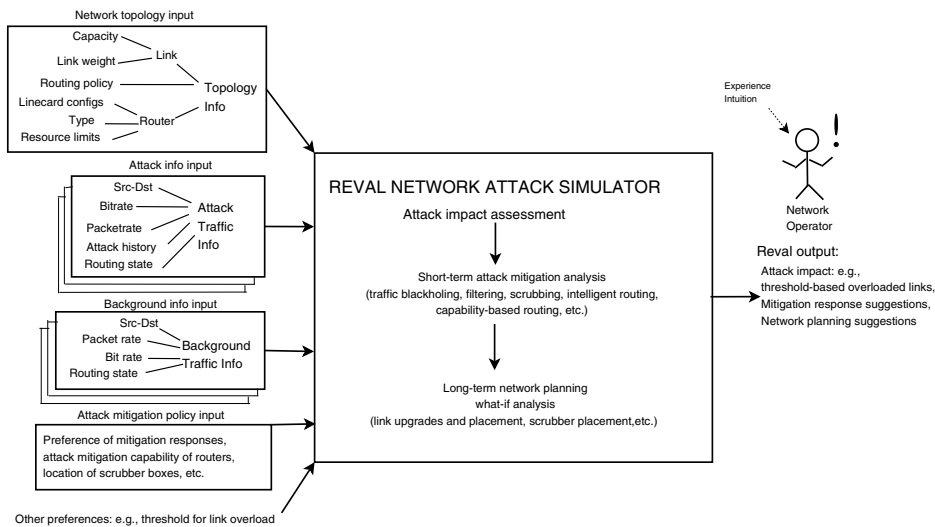


Figure 1: Reval architecture.

the whole network. All this data can be collected from the network and processed in real time while information that changes infrequently, like the network topology, can be updated as required.

2.2 Implementation

In this section, we describe how we build our tool to realize the above goals and capture attack impact at the flow-level. First, we highlight the core functionality of Reval and then discuss various features of the simulator that make Reval an attractive tool for network operators.

2.2.1 Core functionality

We now describe how the information in Figure 1 is used within Reval. The simulator reads in the **topology information** in terms of links and routers and constructs a graph in the Stanford GraphBase [10] format that provides an efficient and well-abstracted representation. We store all additional information of network elements where appropriate by suitably adding to the SGB library data structure. Router types and line card configurations are stored at a per-vertex level while link information of capacity and link weights are stored at a per-arc level. For each network element, resource limits are also stored if applicable.

The **attack information** shown in Figure 1 comprise of multiple traffic flows which together form the ingress-router-interface to egress-router-interface attack traffic matrix. The following procedure is then used in setting up each traffic flow in the matrix: (a) The vertices in the graph corresponding to the ingress and egress routers of a flow are identified; (b) Using the link-state shortest path routing, *e.g.*, as in the OSPF protocol, the shortest path

between the source and destination vertices is found. We have implemented the industry best-practice of splitting traffic flows across equal-cost paths as well; (c) The flow is set up along the shortest path(s) updating the state information of each link carrying the flow. This includes properties such as link utilization, amount of traffic carried, and number of flows carried.

As depicted in Figure 1, the **background traffic information** is also specified as a collection of flows specified at a per-router interface level. These traffic flows together form the background traffic matrix. The procedure for setting up background traffic flows is executed once all attack flows are setup and is similar to that outlined above. At every network element where statistics are updated while setting up a flow, we check whether any limits are violated. If so, the element is said to be overloaded and the background traffic flow currently being set up is said to be impacted. The user of the simulator can specify a threshold of when a network element is considered overloaded. Here, a network element could be something as specific as a line card in a router or a backbone link in the core network.

Other input information required is related to the particular use of the simulator like evaluating attack mitigation policies under certain network conditions, and assigning values for parameters of the simulation based on the experimental scenarios desired.

2.2.2 Peripheral functionalities

Routing algorithm support: The default routing algorithm is the shortest path routing which is widely deployed in most IP networks. We also provide implementations of two other routing algorithms that examples of dynamic routing — namely, the max-flow routing algo-

rithm (using ideas from [11]) and a custom load-sensitive routing algorithm.

Defense mechanisms support: We support multiple defense mechanisms that can be enabled via a configurable file-based interface to the simulator. A modified version of the pushback algorithm [12] is supported. In addition, we also provide means of enabling packet filters with varying capabilities across any set of routers. Though we do not simulate traffic flows at the packet-level, this feature is nevertheless useful as it abstracts the inner workings of the filter while maintaining its effect. As we describe in Section 3, we use this filtering ability to realize *selective blackholing of traffic*, in which a subset of routers drop traffic to a particular destination. Selective blackholing is done local to each router.

A third defense feature which we implement is traffic scrubbing. Scrubbers are “boxes” deployed in the network that take in traffic and scrub the “bad” traffic from the “good” traffic [13]. Again, rather than concerning ourselves with the algorithms used for scrubbing, we abstract the effect of these scrubbers. Scrubbers can be enabled at multiple network locations in the simulator. Since scrubbers are physical boxes, there are only a few scrubbers that are deployed in a real network due to cost considerations. An intelligent routing platform such as RCP [14] could be used to realize both the above defenses in a real network.

Reval also provides the right framework and extensibility to evaluate defense mechanisms not yet common in today’s networks. For example, it can calculate the amount of bandwidth required for control traffic in a capability-based DoS defense mechanism [15]. Towards end-host protection from DoS attacks, [16] suggest changes to the Internet architecture. However, the effect of some of these changes might be overridden by security policies of ISP core networks. Reval helps quantify the effects of core network security policies.

Analysis support: One of the useful analysis support functions that Reval provides is the simulation of the hypothetical case when the particular attack under consideration had been larger in terms of volume rate. This is achieved by multiplying the traffic volume by a constant factor called the “scaling factor” and then simulating using this scaled-up attack. In the future, we intend to scale other attack properties to obtain a more comprehensive picture. In our simulations described later, we are particularly interested in computing the greatest scaling factor (GSF) by which an attack can be scaled before overloading a core or access link.

Granularity of Simulations for Impact Assessment: Simulation granularity affects the quality of results. As simulations become more fine-grained, one would expect the accuracy of results obtained to improve and, in the limit model the real network *exactly*. The trade-off

	Large Tier-1 ISP > 60000 nodes, 0.4 million flows
Real-time	11 secs, 50 MB
w/o Vertex Hash Table	400 secs, 49 MB
w/o Dynamic Programming	900 secs, 45 MB
w/o Graph Reduction	> 4 GB
w/o Functional Reuse (for 5 iterations)	11×5=55 secs, 50 MB (versus 4.5+(11-4.5)×5=37 secs)

Table 1: CPU time and memory utilization: demonstrating performance impact of each optimization.

is higher complexity of simulations. We have explored alternative levels of granularity of simulations, one of which simulates the packet-rate limits of network elements. Routers typically have limitations not only in terms of the volume of traffic they can handle but also in the number of packets serviced owing to a fixed number of packet buffers [17]. In fact, operational experience suggests that the number of packets serviced decreases as more advanced security features are enabled in a router (since more work is required in processing each packet). From an offline study using this simulator, we found that typical attacks impact networks not only via their bit rate but also sometimes due to their packet rate. For the example network in the Section 3, we carry out capacity-based simulations only because details regarding the routers used in that network (and associated packet limits) are not publicly available. However, our simulator has the capability to carry out simulations at more than one granularity level and can be extended to other dimensions of granularity as well.

Impact metrics: A sizable number of impact metrics can be enabled at the time of execution of the simulator. Metrics range from network properties like network core link utilization, the GSF, network-geographic distribution of traffic, network throughput *etc.*

Miscellany: The tool is modular, extensible, configurable and robust to input data errors. The simulator also handles network data as input feeds and executes on an automated basis within the tier-1 ISP.

2.3 Simulator optimizations

Below we enumerate the design optimizations of our simulator. Note that the main goal of these optimizations is to reduce simulator runtime. Table 1 presents a summary of the CPU resource utilization for the real-time version of the simulator (highlighted in **bold**) where one iteration of the core functionality is executed (except for the last entry). The table also highlights the contributions provided by the various optimizations. For our simulations, we used a machine with 2 Intel® Xeon™ CPU of 3.60 GHz processor speed each sharing an overall RAM of 4 GB running on the RHEL OS with Linux kernel ver-

sion 2.4.21. Program code, written in C, was compiled using gcc (version 3.2.3) and the “-O 4” flag.

Hash Table of Vertices: Network elements are easily accessed in constant time using indices to the vertex array while network input data like Netflow records typically use names to refer to network elements. To avoid converting retrieval into an $O(|\text{Vertices}|)$ operation, we construct a hash table where each element is hashed using its name as the key while the hash value stored is its array index in the graph. The cost of constructing the hash table is amortized over the simulations.

For our implementation of the hash table, we directly adopt the hash algorithm and implementation provided by [18]. In our use of the hash table, we store the number of the vertex with its name as the hashing key. From Table 1, we see that using this hash table has sacrificed a small amount of memory in return for two orders of magnitude improvement in time.

Pre-computation of Shortest Paths: ISP network topologies do change but typically not over short time scales (unless there are failures). Most changes happen to customer connectivity to the access network while the core and access networks themselves change infrequently. Also, ISP networks are hierarchically organized: customers connect to access networks which funnel data towards a hub or a backbone router before entering the core network. This hierarchy invariably creates multiple equal-cost paths across access router interfaces but the multiplicity is inherent in the ISP network only and is independent of the customers.

Both the above observations led us to take the approach of pre-computing shortest paths across routers within the ISP network. When a particular traffic flow needs to be set up between customers, we first determine the respective access routers these customers connect to. In our data structure for the network, this takes a single lookup only, since customers are modeled as connecting to a single access router in the ISP network (we model each access interface as connecting to a unique customer of the ISP network; this does not however reduce the accuracy of simulation since the effects of multi-homing would be reflected in the traffic matrices input to the simulator). The pre-computed shortest path(s) between the routers is then used to set up the flow. This way, we can avoid repeated shortest path computations. A useful side effect of pre-computation is that we can accurately model equal-cost paths by splitting traffic at each router according to the number of equal-cost paths available at that router. This is in relation to the shortest paths computation natively available in the SGB library.

Dynamic Programming to Compute Equal-Cost Paths: The above description shows the benefits of pre-computing shortest paths. However, the process of computing *all* equal-cost shortest paths between each

node pair within the ISP network can itself be time-consuming. This is especially true as the typical size of an ISP can be on the order of at least several thousand nodes. So, we adapt the dynamic programming-based all-pairs shortest paths algorithm described in [19]. In particular, we modify it to incorporate the computation of all equal-cost shortest paths of a select set of nodes which we discuss in the next bullet point.

Table 1 illustrates the benefit of dynamic programming over the shortest paths computation native to the SGB software code. We observe significant improvement in run-time in return for a modest 5 MB increase in storage.

Graph Reduction: As stated earlier, we assume that every customer connects to a single access router. So, customer traffic at the time of entering and exiting the network has a single choice of next-hop. Hence, for path computation, it is enough if we consider the core and access network of the ISP alone leaving the customers out of the calculations. This reduces the size of the graph we have to consider for our path computation by two orders — from in excess of 60000 nodes (core, access and customer routers) to little more than 600 nodes. The entry in Table 1 pertaining to this optimization illustrates why, more than just a desired optimization, this is required to execute the simulator on off-the-shelf workstations.

Functionality Re-use: Often the same input data is used in multiple runs of a common piece of simulator code while varying just the parameters fed to the simulations. This could either be within a single scenario or across multiple scenarios. In either case, much time is saved by reusing functionality that is required but remains the same across different iterations. For example, one very useful feature to have is to read in the network topology once only while resetting the network state at each network element every time attack impact needs to be computed given a parameter change only (without changes in network topology). The cost savings in not having to re-read the topology more than offsets the cost involved in visiting every vertex and reinitializing its property variables. Furthermore, storing such data in memory does not make the memory usage prohibitive as is evident from Table 1.

An example where the benefit of functional reuse can be observed is the scenario of computing the GSF value before core network overload. This computation requires changing the scaling factor and re-executing the core functionality code of the simulator. The entry in Table 1 provides the expressions which captures the time required in the two cases when this optimization is and is not enabled (at 5 iterations). For a typical attack on a large tier-1 ISP network, the average value of iterations is 5 which results in an execution time of 37 seconds with optimization and 55 seconds without. Note that in the real-world where topologies change by small

Choice of Data Representation: The Stanford GraphBase (SGB) model [10] provides a useful abstraction of network data where each vertex abstracts a network element and each arc a network link. It also provides a convenient mechanism for storing arbitrarily complex data structures required to maintain state local to each vertex and arc. Moreover, the memory usage is optimum owing to a byte array storage of data. We base our implementation ideas on the GT-ITM project [20] in its use of the SGB model for topology modeling.

3 Case Study: Abilene

Based on our experience, network operators facing an attack typically choose a particular mitigation strategy without evaluating all available defense options due to lack of time. With Reval, we provide network operators with a systematic way of evaluating available defense options in a real-time manner. We illustrate this point in this case study by considering two real-time attack mitigation mechanisms: traffic scrubbing and selective blackholing (described in Section 2).

Map of the United States showing the OC-192c network topology. Major cities are marked as nodes: Seattle, Denver, Kansas City, Chicago, Indianapolis, New York, Washington, Atlanta, Houston, Los Angeles, and Sunnyvale. Solid red lines represent the OC-192c network, while dashed black lines represent other network paths. Arrows indicate the direction of traffic flow. A legend at the bottom right shows a solid line for 'AttackOne' and a dashed line for 'AttackTwo'.

generated using the generalized gravity and simple gravity methods discussed in [23, 24]. Since we require traffic matrices at a per access router level, we use the gravity model to interpolate from the PoP-level TMs [23].

3.1 Oversubscription Factor

USENIX Association

PoP-location	Oversubscription Factor
New York City	1.833
Chicago	1.803
Washington D.C.	1.779
Los Angeles	1.508
Seattle	1.5
Rest	<0.5

Table 2: Oversubscription factor values for Abilene.

link from the access routers to the New York backbone router is only $100/1.833$ (≈ 54.55) % utilized by an attack, at least one core link out of New York is overloaded.

To corroborate the qualitative understanding just obtained, we simulate attacks considering just the top 2 oversubscribed PoPs in New York and Chicago. For our simulations, for each of the 2 PoPs, we consider a set of 100 artificially generated, random-target attacks where in each attack: every access router of a PoP sends traffic to occupy 1% of its link capacity to targets chosen randomly from among all access routers outside the current PoP so that every flow traverses at least one core link.

From Figure 3, for the New York PoP, we see that the maximum GSF value required across all 100 attacks is about 50. The utilization given this GSF value is 50% which is clearly less than the expected value of 54.55% calculated from the Oversubscription Factor. This corroborates our earlier understanding. Observe that the same is the case with the Chicago PoP. From the starting value of both plots, we gather that there are attacks originating from the New York and Chicago PoP that require to be scaled in volume by only 27 and 15 respectively to overload at least one core link. These minima points correspond to attacks whose targets are such that all attack flows traverse a single core link.

3.2 Mitigation Evaluation

In the previous subsection we performed an analysis of the critical points in the Abilene network. As we saw, carefully targeted attacks overload the core at fairly low scaling factor values, meaning that only a part of access link capacity is required for this. Network operators need to employ mitigation mechanisms to lessen or even eliminate attack impact on the core. To this end, we now consider a sample real-time attack mitigation policy used by ISPs today. Our main thesis is *not* that this sample policy is the panacea to the attack mitigation problem but that our tool offers a convenient framework to test a desired real-time mitigation policy.

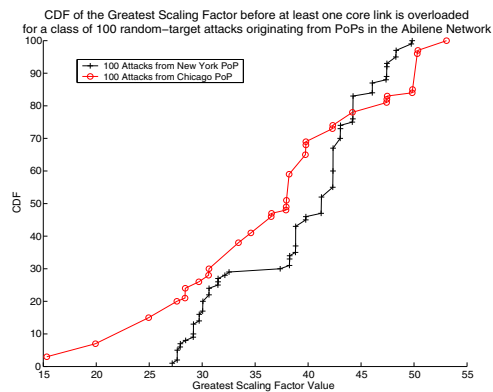


Figure 3: Understanding the strength of an attack originating from the New York and Chicago PoPs required to overload at least one of the respective oversubscribed core links.

3.2.1 A Sample Mitigation Policy

The sample mitigation policy we consider involves a combination of traffic scrubbing and selective blackholing to realize incremental attack mitigation. We aim to answer the following two important questions: given an attack, what is the minimum amount of mitigation required, using a combination of traffic scrubbing and blackholing to:

- *avoid overloading network core?*
- *avoid overloading the target(s) of the attack?*

A flowchart of the policy implementation is shown in Figure 4. A detailed explanation follows. Note that in practice, the steps in the policy are executed until the desired level of mitigation is reached. In addition to the policy, we also consider the mitigation offered by selective blackholing *without* traffic scrubbing for comparison purposes. In this case study, we perform deployment evaluation since we consider particular scrubber deployment scenarios and evaluate this restricted set of options. However, the simulator has the capability of choosing various deployments by itself, in a brute-force manner, and evaluating mitigation options offered in each case.

Step 1 - Ordering attack links: The intuition is that mitigating traffic from links considered in descending order of bit rate offers the best possibility for significant reduction of attack impact.

Step 2 - Traffic Scrubbing: Traffic scrubbers are typically deployed at select locations in the network only. This is because firstly these boxes are expensive, and secondly, their false positive rate is low for large traffic volumes. Scrubbers in general seldom affect genuine traffic resulting in little, if any, collateral damage. At the same time, to realize the same scope of mitigation as blackholing we would need to deploy scrubbers at every router which is expensive. (Henceforth, we refer to “link considered for scrubbing” as “scrubbed link”.)

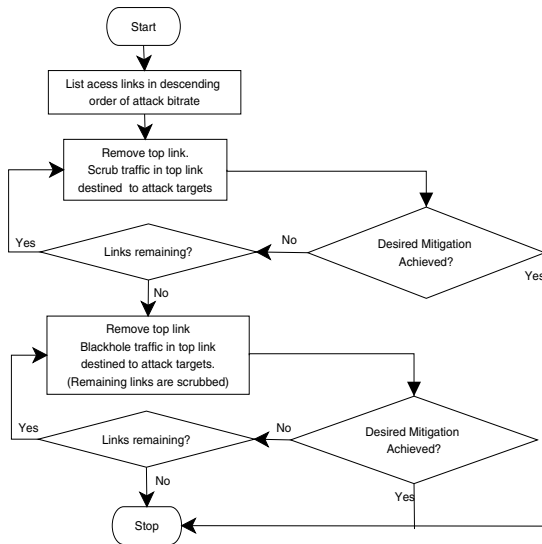


Figure 4: Flowchart of the implementation of our sample mitigation policy involving traffic scrubbing and selective blackholing.

In this step of the policy, given information on the location of the scrubbers, the traffic on a scrubbed link is sent to the closest scrubber in terms of minimum cost paths. This is done by performing a lookup in the pre-computed shortest paths table (explained earlier in Section 2) for each scrubber location and identifying the scrubber with the smallest path cost. Once traffic is scrubbed, residual traffic, if any, is then routed to the original destination. This implementation of scrubbing is reasonable and employed in networks today. An important point is that after identifying the attack target(s), *all* traffic in the scrubbed links need to be sent to the scrubber.

Step 3 - Selective Blackholing with Traffic Scrubbing: Selective blackholing attack traffic provides greater mitigation than traffic scrubbing as all traffic to a set of targets is dropped local to the ingress router without entering the ISP network. However, unlike traffic scrubbing, there is no residual traffic after blackholing; thus the collateral damage is absolute. Hence, *a network operator resorts to blackholing traffic only if required*. Traffic that is not blackholed is scrubbed. (We refer to “link considered for blackholing” as “blackholed link”.)

In our policy implementation, once all attack links are scrubbed, traffic on the attack links destined for the attack targets are blackholed in the same descending order as before. Again, we do not distinguish between good and bad traffic. The higher collateral cost due to blackholing is confirmed from simulations as is demonstrated in Figure 7.

3.2.2 Attack Scenarios

Taking cues from the analysis in the previous subsection, we consider two particular targeted artificial attack scenarios, referred to as **AttackOne** and **AttackTwo**. Both attacks have flow rates such that each attack target’s access router link is just about 95% utilized. The traffic flow rates (in kbps) and the source and target PoPs for these attacks are illustrated in Figure 2. Note that the topology in the figure does not depict access routers while traffic matrices are at the access router level.

We use AttackOne to mainly illustrate the mitigation effects using varying number of scrubbers. AttackTwo on the other hand is a targeted attack aimed at overloading the core links in such a way so that customers on the West Coast of the USA cannot send traffic to those on the East Coast. Despite the large impact of this attack, network cut-off is achieved with a little more than 30% utilization of the access router link capacities! The exact value for the flows are also depicted in Figure 2.

In both attack scenarios, we geographically co-locate scrubbers with backbone routers and so the cost in reaching them is approximately equal to the cost of reaching the co-located backbone router. Also, we assume a scrubbing capability of 0.99 meaning that 99% of all attack traffic entering the scrubber is dropped (this value was chosen from operational experience, and can be adjusted as desired). Another assumption we make is that when a link in the Abilene network (all OC-192s) is 95% or more utilized, it is said to be overloaded (this value is again reasonable and tunable). We use the simulator feature of scaling and analyzing attacks. In particular, we calculate each time the greatest scaling factor (GSF) value by which a particular attack should be scaled in volume such that a core link is overloaded. In cases where the core cannot be overloaded on scaling, we set the GSF to a ceiling value (1000 for AttackOne and 50 for AttackTwo). Even though these two scenarios seem contrived, targeted attacks have the maximum damage capability (as shown earlier) and use of the tool to analyze these worst-case attack scenarios is instructive.

3.2.3 Results

Experiment Using AttackOne

We start with the scenario of deploying one scrubber and progressively add scrubbers at distinct network locations. We then execute the mitigation mechanism presented earlier on these scenarios independently. In all scenarios, we measure the GSF of an attack before overloading at least one core or target link as the case may be, and set it as the Y-axis on the plots. In the simulator, this value is determined by increasing the GSF starting from a low initial value until link overload occurs.

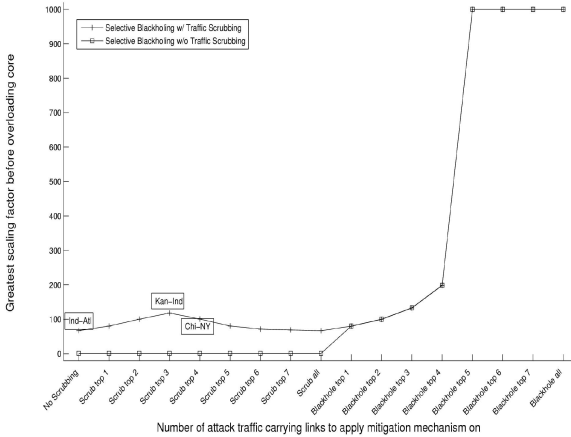


Figure 5: Analysis of effectiveness of mitigation mechanisms on protecting the core network under AttackOne with scrubber at New York. The curve of blackholing without scrubbing is not annotated in the combined plot.

How to interpret the plots? Figure 5 depicts the results from the experiment when a lone scrubber is deployed within the New York PoP under AttackOne. Each point in the plot refers to a particular state of deployment of the mitigation strategy, and this is indicated in the tick labels along X-axis. Since the mitigation deployment we consider is incremental, every successive point denotes the state when one more attack link is scrubbed/blackholed. In the case of the curve 'Selective Blackholing w/o Traffic Scrubbing', no link is scrubbed and this is indicated as a constant zero value in the plots (until the 'Blackhole top' labels). At each state of deployment of the mitigation mechanisms in the network, the plots have been annotated with the set of core edges that are overloaded. So, for instance, at the state when there is no mitigation mechanism (that is, $x=0$), the attack has successfully overloaded the Ind-Atl link while no other core link is overloaded. Since the deployment happens in increments, if successive states have the same set of overloaded core edges, then only the first state is annotated.

Having explained how to read the plots, we now present an analysis. In Figure 5, until the state when the top 3 attack access router links are scrubbed, even though some traffic is diverted away from the Ind-Atl core link towards the scrubber in the New York PoP, there is still sufficient attack traffic to overload the core network. Since traffic through the congested core link decreases with every attack link scrubbed, we need to scale the attacks by a little more to still be able to overload the core. This is indicated by the steady increase in the GSF value. Then, the scrubbing curve begins to drop off as we scrub the top 4 attack links. At this stage, the congestion point is transferred to the Kan-Ind core link as now a large portion of the attack traffic is ferried via the

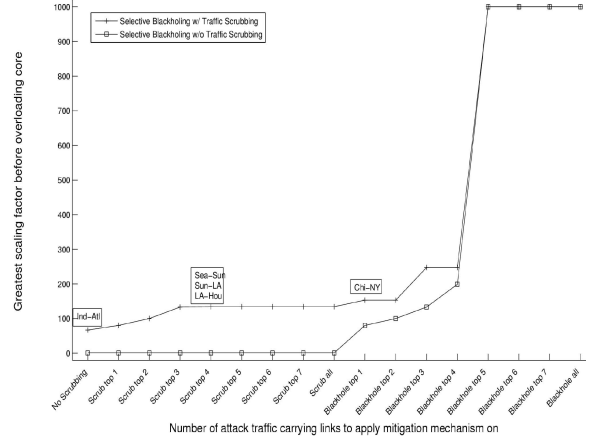


Figure 6: Same scenario as in Fig 5 but with scrubbers at both New York and Houston.

Blackhole top N links	0	1	2	3	4	5	6	7	8
Blackholing Cost ($\times 10^{-3}$)	0	0.8	1.6	3.3	4.2	5.9	7.1	7.1	7.1

Figure 7: Table of costs for the blackholing strategy in terms of percentage of background traffic dropped under AttackOne. The columns corresponds the states starting from "Scrub all" to "Blackhole all" in the above plot.

shortest path from the sources at both Kansas City and Seattle to the scrubber in New York. Observe the trend of steady dip in the scrubbing curve of the GSFs. There is already enough traffic transferred to New York that by the time we scrub some more traffic, due to aggregation the core link leading up to the New York router is overloaded. This is reflected in the plot by the annotation of "Chi-NY". This plot highlights the expected effect of having a single scrubber which becomes the single point of congestion in the network.

Once all attack links are scrubbed, the top attack link is blackholed. Up till when the top 4 attack links are blackholed, blackholing traffic is not sufficient to shift the congestion point from the New York core link. This is indicated by the annotation trend as the GSFs increase. However, when the top 5 attack links are blackholed, the number of attack links still sending attack traffic into the core network is so low that even at full capacity these links together cannot overload any core link in the network. This is indicated by the plateau at the ceiling value 1000.0 at the tail of the blackholing curve. Note that the curve for the blackholing without scrubbing mitigation strategy nearly coincides with the blackholing with scrubbing curve indicating that in this single scrubber situation, the effects of scrubbing are negligible compared to that of blackholing.

Figure 6 shows how the network copes with AttackOne when there are 2 scrubbers deployed at New York and Houston respectively. Since the closest scrubber to

the PoPs on the West Coast is at Houston, traffic from the sources in Seattle and Kansas City are directed towards Houston reducing the load on the New York core links. This is directly observable as the lack of the dip in the scrubbing curve observed in the earlier figure. In this 2-scrubber scenario, the bottleneck links are those along the path from Seattle to Houston which is reflected in the annotation in the plot. Similar to the earlier scenario, as we start blackholing, the only overloaded core link at the GSF is the Chi-NY core link. Also note that since traffic destined for scrubbing is now distributed between 2 scrubber locations not in close network proximity, attack traffic is much more diluted through the network core. So, the attack needs to be scaled up by a higher factor to overload the core.

The best evidence for the benefits of scrubbing at 2 locations is provided by comparing the two blackholing curves in Figure 6. At every state of the network (before $x=5$), the deployment of 2 scrubbers in addition to blackholing has bought the network operator nearly twice as much network capacity than by blackholing only. This can be seen from the two-fold increase in the GSF values of the 2 curves. Remember that the blackholing without scrubbing curve nearly coincided with the blackholing with scrubbing using one scrubber curve. So, these observations serve to compare indirectly between the 2-scrubber and 1-scrubber deployment scenarios.

We considered other simulation scenarios using more scrubbers under the effect of AttackOne. Due to lack of space, we do not present those plots. In essence, when an additional scrubber is added to the network, the attack traffic in the network becomes more diluted. Of course deploying scrubbers right at the backbone router nearest to the sources almost completely eliminates attack traffic when traffic scrubbing is performed and so we analyze only the contrary deployment. In almost all these simulation scenarios, we observed a similar two-fold improvement in effectiveness of using scrubbing along with blackholing as opposed to using blackholing only.

Using the above, the network operator is able to quantify attack impact of AttackOne as well as mitigation offered by the strategies in the sample policy. Therefore in answering the first of the two questions posed earlier, she can for example choose to defend her network by blackholing the top 4 attack links knowing that the attack now needs to ramp up its volume by more than 200 to be able to still overload the core. Table 7 gives the costs involved in blackholing background traffic. Note that these costs are dependent only on the traffic matrices and are independent of traffic scrubbing. The percentage of background traffic dropped due to blackholing versus total background traffic is the measure of cost captured in the table. Clearly, there is an increase in the cost paid by the network operator in terms of collateral damage as

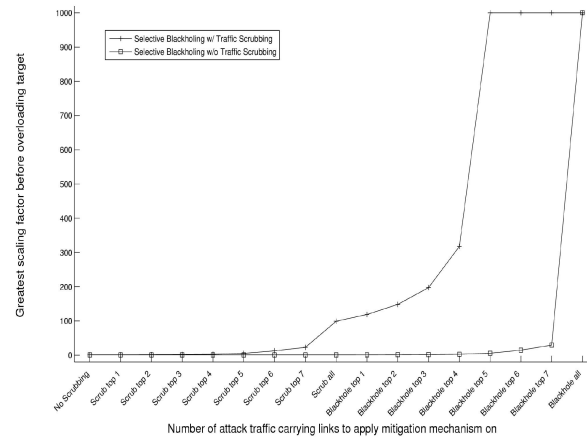
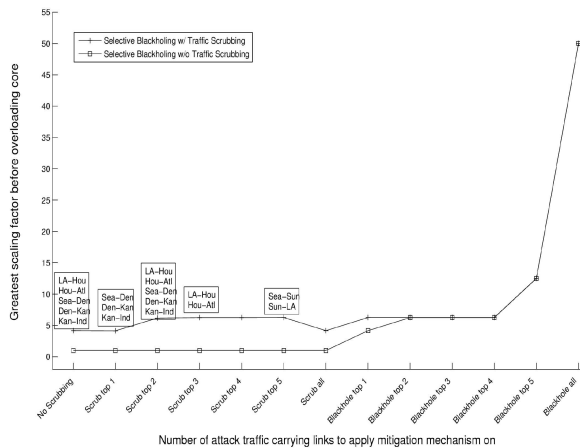


Figure 8: Analysis of effectiveness of mitigation mechanisms on protecting target links under AttackOne.

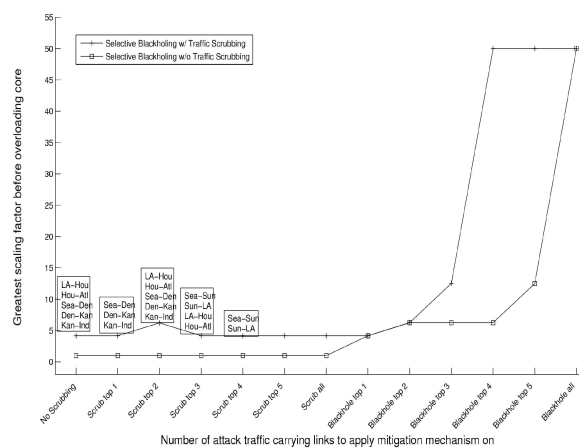
more access links are considered for blackholing. So, she needs to base her final mitigation decision on both the strength of mitigation desired as well as the cost she is willing to pay.

Figure 8 represents a similar plot as before but with the focus now on avoiding overloading the target link. Recall that the attack at scaling factor of 1 just about overloaded the target access link. The plot depicts the scenario when a single scrubber is deployed in New York. Under the other scrubber deployment scenarios considered earlier, we obtained exactly the same numbers indicating that as far as protecting the target is concerned, scrubber deployment location does not matter (as long as the network core itself is not affected). On comparing the GSFs to overload the core versus that to overload the target link for the same network states, we observe that the latter GSFs are much smaller as expected.

From the curves, it becomes clear that scrubbing is more effective in protecting the target(s) than protecting the core. For a combination of blackholing and scrubbing under AttackOne, Reval is able to provide two benefits simultaneously. First, multiple deployment options are provided based on the level of defense desired. Secondly, for each mitigation option, Reval quantifies the increased attack capacity that the network can now sustain before again overloading the core. Using scrubbing alone, at the very best the network operator can hope to reduce the traffic reaching the target link(s) by a maximum factor of $\frac{1}{1-f}$ where f is the scrubbing accuracy. In our case, this theoretical maximal defense value is $\frac{1}{1-0.99}=100$. The actual value is dictated by when the core network is overloaded due to the diversion in traffic. Reval can provide the network operator with the exact numbers within this maximal range as just illustrated.



(a) With scrubber in Seattle



(b) With scrubber in LA

Figure 9: Choice of scrubber deployment under AttackTwo.

Experiment Using AttackTwo

AttackTwo as described earlier was designed to completely cut off coast-to-coast communication requiring surprisingly small bandwidth to do so. Exploring the full space of mitigation options is a cumbersome task even after restricting ourselves to the two strategies of traffic scrubbing and blackholing. Instead, we ask a rather qualified question: given the availability of one scrubber, where should it be placed so that traffic scrubbing and, if necessary, selective blackholing give the network operator maximum leeway in dealing with the attack?

Since the attack partitions the network into two and all sources of the attack are on the West Coast, it makes sense to deploy the scrubbers in one of the West Coast PoPs. Also, since the sources of attack are at Seattle and LA, deploying the scrubber at other West Coast nodes does not buy the network operator much (this is substantiated by results from the simulator not shown here). Figure 9 then plots the results of the simulator under the two scenarios - scrubber deployed in Seattle and LA respectively. Observe the plots for the GSF values for corresponding states especially paying attention to the blackholing with scrubbing curves. Once the top 3 attack links are considered for blackholing, the effect of scrubbing coupled with the choice of the flows to scrub make LA an attractive choice for scrubber deployment. Also, notice that to protect the core under this attack, with the scrubber deployed at LA, it is enough to blackhole traffic from 4 out of the 6 attack access links while in the case of the deployment at Seattle all 6 attack links need to be considered for blackholing. Thus on both counts, we have LA as the better location for scrubber deployment.

With additional analysis, we discovered the existence of an interesting phenomenon that is brought out by the simulator: in our mitigation strategy, we pick the top attack links starting from the highest. However, a tie break-

ing mechanism is required to deterministically decide between equally voluminous attack links. We see why this is important now: in AttackTwo since there are 6 attack traffic flows, 3 each from Seattle and LA, all of the same attack volume, there is already a tie. To break the tie, suppose we choose Seattle's attack links before considering LA's attack links, and we consider the deployment of the scrubber at LA first. As a result, because of the order in which we choose to scrub/blackhole attack traffic, we only need to blackhole the top 4 attack links. These correspond to the 3 sources in Seattle (thus relieving the load on the Kan-Ind core link) and one source in LA. The remaining 2 sources are in turn considered for scrubbing performed locally within the LA PoP. On the other hand, when the scrubber is deployed at Seattle, after blackholing the top 4 attack links, we need to send all traffic from 2 attack links in LA to Seattle to be scrubbed. This results in an overload of all core links in the path from LA to Seattle thus requiring to blackhole all 6 attack links in order to protect the core.

Note that by using a purely qualitative study based on the AttackTwo traffic matrix, we are led to believe that both LA and Seattle are equally significant PoPs and deployment at either location yields the same benefit! Such an ad-hoc decision does not factor in practical aspects of mitigation mechanism implementation. This particularly becomes important if the network operator assigns greater priority to traffic from LA than Seattle owing to greater profit potential. Even though equal traffic rate attacks are rare in the real world, the qualitative take-away from this scenario holds. Thus, the network operator using Reval can gain knowledge of both the **effectiveness** and **real-world implementation issues** of the desired mitigation policy.

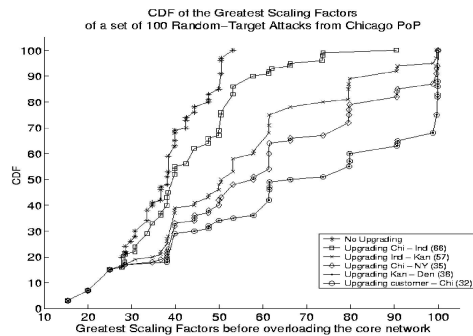


Figure 10: Link Upgrade experiment for a class of 100 random-targeted attacks originating from the Chicago PoP. In the legend, along with the particular link being upgraded, we also present the number of attacks out of the set that overload that link.

3.3 Long-Term Network Planning: Capacity Upgrading

In this subsection, we illustrate a use for our tool in helping a network operator carry out effective long-term network planning. We specifically use the tool to shed light on which links in the network require upgrading to higher capacities. For this, we first choose to upgrade one at a time only the core links that are impacted frequently by attacks (and these are oversubscribed core links). After each link upgrade, all attacks are simulated again to identify the next most frequently impacted link, and so on. Secondly, we upgrade, if at all, a link to have four times its current capacity. This is an industrial practice that amortizes equipment cost over future demands.

Figure 10 shows the result of the link upgrade experiment carried out using the class of 100 attacks from the Chicago PoP described earlier in Section 3.1. We only consider a maximum scaling factor of 100 in the graph without loss of generality. As expected, the most frequently overloaded link is the core link involving the Chicago backbone router. It is interesting to note that on a single link upgrade, nearly 10 attacks out of the 100 required to be scaled by 25 more than before. Also, even after upgrading all core links, more than 60% of the attacks have a GSF value smaller than 100. Another observation is that there is a sizable number of attacks that manage to overload a core link at low scaling factors (15% of attacks have $GSF \leq 25$). Since these overloaded core links do not occur frequently, they are never upgraded. This illustrates the use of Reval to quantitatively assess which points in the network require upgrading and how much respite from traffic load these link upgrades provide. The analysis also reflects the choices and compromises network operators make in choosing to defend against the common case attacks while still being susceptible to isolated instances.

4 Experience with a Large Tier-1 ISP Network

In this section we present results from evaluating the mitigation strategies of traffic scrubbing and selective blackholing on real data obtained from a large tier-1 ISP. We describe our experience of using the tool over a period of one month. Both background traffic and network topology information are obtained in the form of real-time feeds — the former as Netflow records while the latter as router configuration files. The attack input data for the simulator is obtained from alarms provided by the commercial DDoS detection system. Then these alarms are used to obtain relevant Netflow traffic records from a subset of routers producing the attack traffic matrix. In our network, the commercial detection system can be configured to provide these statistics in real time. Attack information could also be obtained by using real-time anomaly detection systems like [25]. Flow properties required for the simulator in terms of source, target, bit rate *etc* are directly obtained from Netflow.

As mentioned in Section 2, we have automated the execution of our simulator on real data obtained from the tier-1 ISP. Restating the performance numbers, the execution of the core functionality of the simulator on the ISP network of size in excess of 60000 nodes with about 0.4 million traffic flows took 11 seconds approximately.

We considered the simulation scenario of analyzing the effectiveness of the mitigation policy of the previous section with a slight modification. We attempt to analyze the decision a network operator would take in the real world using results from the simulator. The data for this simulation scenario comprised of 144 attacks (as classified by the commercial DDoS detection system) that occurred over the month of September in 2005. For each attack, we obtained statistics on the GSF value before overloading the core under the network state at the time of the attack. This was done for various mitigation deployment states in the two strategies of traffic scrubbing and selective blackholing with traffic scrubbing.

None of the attacks overloaded any core links in their original form, and therefore needed to be scaled up. From preliminary analysis, we found that 77 of the 144 attacks did not have enough volume rate to overload the core. That is, even with maximum scaling (when all ingress access links of an attack are used to their full capacity), these attacks could not individually overload any core link. Also, about 24 of the attacks pertained to instances when the network data collected was not reliable. For the remaining 43 attacks, we fixed a particular scaling factor value for which all of these attacks would overload at least one core link. With this as the base, we then obtained the best mitigation strategy required to avoid overloading the core. By “best”, we refer to the

strategy that required the least amount of blackholing as opposed to traffic scrubbing since blackholing results in greater collateral damage.

The number of unique responses offered by the simulator as the best mitigation strategy for these 43 attack instances was 16. Each involved blackholing and/or scrubbing at least one unique network ingress link not considered in any other instance. Thus, Reval revealed that mitigation against attacks in the real-world involves a constantly changing region of deployment of the mitigation mechanisms rather than a fixed deployment. Moreover, the region of deployment is dictated by the nature of the particular attack under consideration.

As further analysis, we investigated the usefulness of the solution by analyzing the scenario of “How much impact would the core network suffer if the second best mitigation strategy was chosen instead of the best?” For these 43 attacks, we found that the second best mitigation strategy overloaded at least one core link for 30 attacks and in a couple of instances, the attack managed to overload 2 core links while the remaining did not overload any core links. So not only did Reval provide the mitigation option that was best in terms of effectiveness but was also better by a fair margin in a large number of attacks.

5 Related Work

The EMIST project [26], a sister project of the DETER test bed, aims to provide a framework and methodology for studying particular classes of network attacks and defense mechanisms. Using a test bed limits the scale of experimentation. Also, defense mechanisms need to be deployed on the test bed either by procuring the right hardware like in the case of traffic scrubbers or implementing functional software. Such test beds are more useful in understanding network attacks in an *off-line* setting unlike our tool.

ns2 [27] is a popular network simulator that provides capabilities for packet-level simulations. However, emphasis is on lower-layer simulations rendering it unsuitable to large networks. Also, neither *ns* by itself nor any third-party code address security simulations. Likewise, GloMoSim [28] is a discrete-event mobile network simulator that is currently not used in security research. SSFNet [29] has a similar scalability to that of *ns2* [30]. MicroGrid [31] is a network simulator built to realize realistic large scale online simulations. An associated project is the ModelNet network emulator [32] which also focuses on scalable network simulations. In both works, the goal of the system is to gauge distributed application performance under different network load scenarios, and are not suited for *network* attack impact and mitigation studies.

There have been studies on building simulators adopting a particular model of network traffic. However, almost all these models assume an underlying working theory on patterns in traffic properties. It is not clear how these theoretical traffic models apply to network attack traffic that are inherently non-uniform. Related work in this regard is the network calculus-based approach to scalable network simulations [33]. Other work include the network simulator presented in [34] where a fluid-based model is used to make packet-level simulations more scalable. However, this simulator when run on large IP networks is still far from real time. Moreover fluid models are not viable for light or sporadic traffic.

Parallel to the research on models for faster simulation, there has been considerable work on providing techniques that aim to increase speed of execution in a simulator-independent manner. We mention few of these techniques here while noting that these techniques are complimentary to our simulator. A recent approach to scalable fast large scale simulations is to identify network invariants, preserve them in scaling down simulations and accordingly scaling up the results obtained [35] thus vastly improving on a full-blown packet-level simulation. Another related approach is in the world of building a parallel network simulator with the idea of intelligently parallelizing simulations at a part level and then communicating once in a while to realize system-wide simulations [36].

6 Conclusion

We have presented the design and implementation of *Reval* — an operational support tool that quantifies the impact of attacks on any-sized networks, analyzes the effectiveness of mitigation strategies, and provides comprehensive answers pertaining to a wide range of mitigation deployment scenarios, all in a real-time manner. Moreover, the use of *Reval* brings to the attention of the network operator potential real-world deployment issues of desired mitigation strategies otherwise observable only after practical experience. Using live data from a tier-1 ISP network of size in excess of 60000 nodes with close to 0.4 million traffic flows simultaneously, *Reval* executes in about 11 seconds. Given a sample mitigation policy in this real network, *Reval* identifies the most effect mitigation option after evaluating each option in about 35 seconds.

Though *Reval* was discussed from the point of view of network attacks in this paper, it could readily be used in a host of network scenarios: analyze network anomalies like flash crowds, routing-driven traffic surges, worm propagation; understand attack impact and mitigation from perspective of customers of the ISP; strengthen the network via capacity planning; study network topology

design using various graph theoretic and other parameters; compare and contrast defense mitigation strategies; quantify the conditions for particular network-level mechanisms like new routing algorithms, defense algorithms *etc.* to work successfully.

It behooves us to state that the simulator we have built is by no means complete or exhaustive in terms of functionalities desired by security researchers. We are currently looking to improve the simulator along a few dimensions including support for packet-level simulations, simulation time, and accurate cost metrics. Reval could be executed on latest network state by incorporating live data feeds that relay changes in the network like IGP changes. Even though Reval evaluates a host of mitigation strategies, choosing the particular strategy is a manual decision taken after making various trade-offs including the level of collateral damage. These trade-offs make the process of attack defense complicated, and difficult to eliminate manual intervention. Nevertheless, Reval is a first step towards its automation.

Acknowledgements

We thank Mukesh Agrawal, Patrick Verkaik, Adrian Cепену, the anonymous reviewers and our shepherd Geoff Voelker for the many criticisms and suggestions that have helped shape this paper.

References

- [1] R. Richmond, "Firms Join Forces Against Hackers", *Wall Street Journal*, March 28, 2005.
- [2] D. Pappalardo and E. Messmer, "Extortion via DDoS on the Rise", <http://www.networkworld.com/news/2005/051605-ddos-extortion.html>, May 16, 2005.
- [3] A. Hussain, J. Heidemann, and C. Papadopoulos, "A Framework for Classifying Denial of Service Attacks", in *Proc. ACM SIGCOMM*, 2003.
- [4] P. Barford, J. Kline, D. Plonka, and A. Ron, "A Signal Analysis of Network Traffic Anomalies", in *Proc. ACM SIGCOMM Workshop on Internet Measurement*, 2002.
- [5] R. Malan, F. Jahanian, J. Arnold, M. Smart, P. Howell, R. Dwarshius, J. Ogden, and J. Poland, "Observations and Experiences Tracking Denial-Of-Service Attacks Across a Large Regional ISP", <http://www.arbornetworks.com/downloads/research37/nanogSlides4.pdf>, 2001.
- [6] P. Vixie, G. Sneeringer, and M. Schleifer, "Events of 21-Oct-2002", <http://d.root-servers.org/october21.txt>, 2002.
- [7] D. Moore, G. Voelker, and S. Savage, "Inferring Internet Denial of Service Activity", in *Proc. USENIX Security Symposium*, 2001.
- [8] "Arbor networks", <http://www.arbornetworks.com/>.
- [9] "Mazu networks", <http://www.mazunetworks.com/>.
- [10] D. Knuth, "The Stanford GraphBase: A Platform for Combinatorial Computing", Addison-Wesley, 1994.
- [11] B. Cherkassky and A. Goldberg, "On Implementing Push-Relabel Method for Maximum Flow Problem", *Algorithmica*, vol. 19, pp. 390–410, 1997.
- [12] R. Mahajan, S. Bellovin, S. Floyd, J. Ionnadis, V. Paxson, and S. Shenker, "Controlling High Bandwidth Aggregates in the Network", in *ACM CCR*, 2002, vol. 32:3, pp. 62–73.
- [13] "Cisco Anomaly Guard Module", <http://www.cisco.com/>.
- [14] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and implementation of a Routing Control Platform", in *Proc. NSDI*, 2005.
- [15] T. Anderson, T. Roscoe, and D. Wetherall, "Preventing internet denial-of-service with capabilities", in *Proc. Hotnets-II*, 2003.
- [16] A. Greenhalgh, M. Handley, and F. Huici, "Using Routing and Tunneling to Combat DoS Attacks", in *Proc. SRUTI*, 2005.
- [17] "Cisco 12000 Series Internet Router Architecture: Packet Switching", <http://www.cisco.com/>.
- [18] B. Jenkins, "A Hash Function for Hash Table Lookup", <http://burtleburtle.net/bob/hash/doobs.html>, 1997.
- [19] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, 2nd edition, 2001.
- [20] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to Model an Internetwork", in *Proc IEEE INFOCOM*, 1996.
- [21] "Abilene network", <http://www.internet2.edu/abilene>.
- [22] L. Li, D. Alderson, W. Willinger, and J. Doyle, "A First-Principles Approach to Understanding the Internet's Router-level Topology", in *Proc. ACM SIGCOMM*, 2004.
- [23] Y. Zhang, M. Roughan, C. Lund, and D. Donoho, "An Information-Theoretic Approach to Traffic Matrix Estimation", in *Proc. ACM SIGCOMM*, 2003.
- [24] Y. Zhang, M. Roughan, N. Duffield, and A. Greenberg, "Fast Accurate Computation of Large-Scale IP Traffic Matrices from Link Loads", in *Proc ACM SIGMETRICS*, 2003.
- [25] V. Sekar and N. Duffield and J. van der Merwe and O. Spatscheck and H. Zhang, "LADS: Large-scale Automated DDoS Detection System", in *Proc. USENIX*, 2006.
- [26] "Deter/Emist Project", <http://www.isi.edu/deter/projects.html>.
- [27] "ns-2", <http://www.isi.edu/nsnam/ns/>.
- [28] "GloMoSim", <http://pcl.cs.ucla.edu/projects/glomosisim/>.
- [29] "SSFNet", <http://www.ssfnet.org>.
- [30] D. Nicol, "Comparison of Network Simulators Revisited", <http://www.ssfnet.org/Exchange/gallery/dumbbell/dumbbell-performance-May02.pdf>, 2002.
- [31] X. Liu and A. Chien, "Realistic Large Scale Online Network Simulation", in *Proc. ACM Conf. on High Performance Computing and Networking*, 2004.
- [32] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker, "Scalability and Accuracy in a Large-Scale Network Emulator", in *Proc. OSDI*, 2002.
- [33] H. Kim and J. Hou, "Network Calculus Based Simulation for TCP Congestion Control: Theorems, Implementation and Evaluation", in *Proc. IEEE INFOCOM*, 2004.
- [34] Y. Liu, V. Misra F. Presti, D. Towsley, and Y. Gu, "Fluid Models and Solutions for Large Scale IP Networks", in *Proc. ACM SIGMETRICS*, 2003.
- [35] H. Kim, H. Lim, and J. Hou, "Accelerating Simulation of Large-Scale IP Networks: A Network Invariant Preserving Approach", in *Proc. IEEE INFOCOM*, 2006.
- [36] Y. Liu B. Szymanski, A. Sastry, and K. Madnani, "Real-Time On-Line Network Simulation", in *Proc. IEEE Intl. Workshop on Distributed Systems and Real-Time Applications*, 2001.

LADS: Large-scale Automated DDoS detection System

Vyas Sekar
Carnegie Mellon University

Nick Duffield
AT&T Labs-Research

Oliver Spatscheck
AT&T Labs-Research

Jacobus van der Merwe
AT&T Labs-Research

Hui Zhang
Carnegie Mellon University

Abstract

Many Denial of Service attacks use brute-force bandwidth flooding of intended victims. Such volume-based attacks aggregate at a target's access router, suggesting that (i) detection and mitigation are best done by providers in their networks; and (ii) attacks are most readily detectable at access routers, where their impact is strongest. In-network detection presents a tension between scalability and accuracy. Specifically, accuracy of detection dictates fine grained traffic monitoring, but performing such monitoring for the tens or hundreds of thousands of access interfaces in a large provider network presents serious scalability issues. We investigate the design space for in-network DDoS detection and propose a triggered, multi-stage approach that addresses both scalability and accuracy. Our contribution is the design and implementation of LADS (**L**arge-scale **A**utomated **DDoS** detection **S**ystem). The attractiveness of this system lies in the fact that it makes use of data that is readily available to an ISP, namely, SNMP and Netflow feeds from routers, without dependence on proprietary hardware solutions. We report our experiences using LADS to detect DDoS attacks in a tier-1 ISP.

1 Introduction

The last few years have seen a steady rise in the occurrence and sophistication of distributed denial of service (DDoS) attacks. Armies of botnets comprised of compromised hosts can be utilized to launch attacks against specific Internet users such as enterprises, campuses, web servers, and homes. In this paper, we focus on an important class of DDoS attacks, namely, brute force flooding attacks. We observe that access links are typically the bottlenecks for most Internet users, and that attackers can easily send sufficient traffic to exhaust an user's access link capacity or overload the packet handling capacity of routers on either end of the link [9].

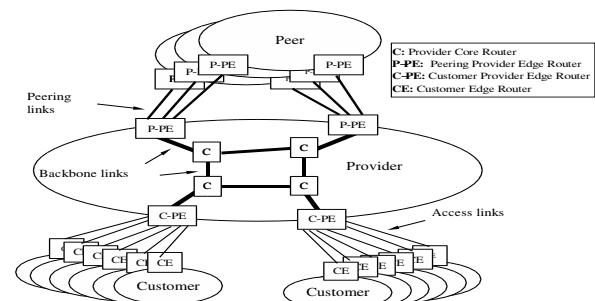


Figure 1: Components of a Provider Network

Brute force flooding attacks are easy for attackers to launch but are difficult for targeted users to defend, and therefore represent a threat to Internet users and services. Given the limited capacity of most access links on the Internet, a successful DDoS attack needs to involve a relatively small number of attack sources. In addition, the size of some reported botnets [27] suggests that a determined attacker might be capable of overloading even the largest access links. From a user's perspective, a bandwidth attack means its in-bound capacity is exhausted by incoming attack traffic. Given that a user often controls only one end of the access link, for example via a Customer Equipment or CE router (see Figure 1), while its ISP controls the other end (referred to as C-PE, or Customer-Provider Edge router), once an access link is overloaded there is precious little that the target of the attack can do without the assistance of its ISP. In fact, automated DDoS detection and mitigation mechanisms originating from the customer side of an access link become useless once the access link itself is overloaded.

For such brute-force bandwidth attacks, we reason that a very promising architecture is one that performs *in-network detection and mitigation*. While it is possible for individual customer networks to deploy detection mechanisms themselves, several practical constraints arise. Small to medium-sized enterprise customers typically possess neither the infrastructure nor the operational ex-

expertise to detect attacks, as it is not cost-effective for them to deploy and manage dedicated monitoring capabilities. Also, the customers' limited resources (human and network) are already substantially overwhelmed during DDoS attacks. Further, detecting attacks at the edge will have little or no effect unless the upstream provider takes appropriate actions for attack mitigation. Network providers on the other hand already possess the monitoring infrastructure to observe and detect attacks as they unfold. Also, since it is upstream of users' access links, an ISP can help customers defend against bandwidth attacks by deploying appropriate filtering rules at routers, or alternatively using routing mechanisms to filter packets through scrubbers [8] to drop malicious packets. In this paper, we focus on in-network detection of DDoS attacks. The challenge is to come up with a solution that satisfies multiple competing goals of scalability, accuracy, and cost-effectiveness.

We propose a triggered, multi-stage infrastructure for detection and diagnosis of large-scale network attacks. Conceptually, the initial stages consist of low cost anomaly detection mechanisms that provide information to traffic collectors and analyzers to reduce the search space for further traffic analysis. Successive stages of the triggered framework, invoked on-demand and therefore much less frequently, then operate on data streams of progressively increasing granularity (e.g., flow or packet header traces), and perform analysis of increasing computational cost and complexity. This architecture fits well with the hierarchical and distributed nature of the network. The early stages require processing capabilities simple enough to be implemented in a distributed fashion for all customer-facing interfaces. The later, more sophisticated, processing capabilities can be more centralized and can thus be shared by many edge routers.

We have designed and implemented an operational DDoS detection system called LADS, based on this triggered multi-stage architecture, within a tier-1 ISP. Our system makes use of two sources of data: SNMP and Netflow, both of which are readily available in commercial routers today. We adopt a two-stage approach in LADS. In the first stage, we detect volume anomalies using low-cost SNMP data feeds (e.g., packets per second counters). These anomalies are then used to trigger flow-collectors that obtain Netflow records for the appropriate routers, interfaces, and time periods. We then perform automated analysis of the flow records, using uni-dimensional aggregation and clustering techniques, to generate alarm reports for network operators.

There are several natural advantages to our approach, in terms of deployment cost, detection coverage, and manageability. Providers incur little or no additional deployment and management costs, because we use data sources that are readily available, and the infrastructure

to instrument and manage the data feeds is typically already in place. Our system provides a low-cost solution for ubiquitous deployment across thousands of customer interfaces, as it does not rely on proprietary hardware solutions. In order to minimize the number of hardware monitoring devices, and hence cost, providers deploy commercial monitoring solutions at selective locations in the network (for example, in the core and at peering edges). Such an infrastructure is likely to miss smaller attacks which, while large relative to the targeted interface, are small amongst aggregate traffic volumes in the core. In contrast, our system has ubiquitous monitoring but no additional cost, and can perform anomaly detection considering both traffic volume and link speed for all customer-facing interfaces.

2 Related Work

The spectrum of anomaly detection techniques ranges from time-series forecasting (e.g., [5, 26]) and signal processing (e.g., [4]), to network-wide approaches for detecting and diagnosing network anomalies (e.g., [19, 34]). These approaches are intended for detecting coarse-grained anomalies, which are suitable for use as initial stages in a triggered approach for scalable DDoS detection.

Also related to our multi-stage approach are techniques for fine grained traffic analysis. These include techniques for performing detailed multi-dimensional clustering [11, 32] and solutions for online detection of heavy-hitters and attacks using counting algorithms and data structures [12, 17, 35].

Moore et al. [24] observed that many types of attacks generate backscatter traffic unintentionally. Network telescopes and honeypots [30] have also been used to track botnet and scan activity. Some early DoS attacks used source address spoofing to hide the sources of the attacks, and this motivated work on IP traceback (e.g., [6, 28, 29]).

There are several commercial DDoS detection systems (e.g., [2, 22]) available today. Since these rely on proprietary hardware and algorithms, we cannot evaluate the differences between the algorithms used in LADS and these commercial systems. There are, however, two qualitative architectural advantages of LADS over these systems. The first issue is one of deployment cost. To provide diagnostic capabilities and detection coverage across all customer interfaces, similar to LADS, providers would have to deploy proprietary hardware devices covering every customer-facing interface and thus incur very high deployment costs. In contrast, LADS uses existing measurement feeds, providing a significant reduction in deployment and management cost for providers. The second issue is one of scale – dealing

with large-scale data feeds from thousands of network monitors. We use a triggered approach to scale down the collection and computation requirements of large-scale attack investigation. We are not aware of any existing work or commercial product which addresses problems at this scale.

Solutions for mitigating DDoS attacks often rely on infrastructure support for either upstream filtering (e.g., [21]), or use network overlays (e.g., [16]). Capabilities-based approaches (e.g., [33]) focus on re-designing network elements to prevent flooding attacks. End-system solutions for handling attacks combine Turing tests and admission control mechanisms (e.g., [15, 25]) to deal with DDoS attacks. Reval [31] helps network operators to evaluate the impact of DDoS attacks and identify feasible mitigation strategies in real-time. Mirkovic and Reiher [23] provide an excellent taxonomy of DDoS attacks and defenses.

The use of triggers for scalable distributed traffic measurement and monitoring has been suggested in AT-MEN [18] and by Jain et al. [13]. While scalability using triggered monitoring is a common theme, our contribution is the use of triggered framework for DDoS detection using heterogeneous data sources.

3 Scalable In-Network DDoS Detection

Having argued for the necessity of in-network DDoS detection (and mitigation), we now consider the implications of this approach for building a detection system in a large provider network. Like any anomaly detection system the main requirement is accuracy, i.e., having a low false alarm and miss rate. The second requirement is timeliness: to be of practical value a detection system should provide near real time detection of attacks to allow mitigation mechanisms to be applied. Third, the system should cover all (or most) customers of a provider. The number of customers could range from a few hundred for small providers to hundreds of thousands for very large providers.

These requirements have significant system scalability implications: (i) Is it feasible to collect information that is detailed enough to allow attack detection on a per-customer basis? (ii) Is it feasible to perform in timely fashion the processing involved with the detection on a per-customer basis?

3.1 Triggered Multistage DDoS Detection

There are two sources of complexity for large-scale attack detection and diagnosis: *Collection* and *Computation*. The collection complexity arises from the fact that data streams have to be selected from monitoring points

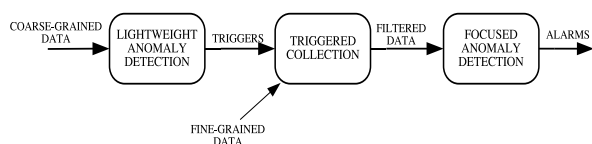


Figure 2: Triggered Multistage DDoS Detection

(links/routers), and either transported to an analysis engine (possibly centralized) or provided as input to local detection modules. The computation complexity arises from the algorithms for analyzing the collected data, and the sheer size of the datasets. We observe that not all detection algorithms have the same complexity: the differences arise both from the type of data streams they operate on and the type of analysis they perform.

Consider two types of data streams that are available from most router implementations: simple traffic volume statistics that are typically transported using SNMP [7], and Netflow-like [3] flow records. Enabling the collection of these two measurements on routers incurs significantly different costs. There are three main cost factors: (i) memory/buffer requirements on routers, (ii) increase in router load due to the monitoring modules, and (iii) bandwidth consumption in transporting data-feeds. SNMP data has coarse granularity, and the typical analysis methods that operate on these are lightweight time-series analysis methods [4, 5, 26]. Flow-level data contains very fine grained information, and as a result is a much larger dataset (in absolute data volume). It is easy to see that the flow data does permit the same kind of volume based analysis that can be done with the SNMP data. However, the flow data is amenable to more powerful and fine-grained traffic analysis [11, 32] which can provide greater diagnostic information.

The presence of heterogeneous data sources which offer varying degrees of diagnostic abilities at different computation and collection costs raises interesting design questions. At one extreme we could envision running sophisticated anomaly detection algorithms on the fine granularity data (i.e., Netflow) on a continuous basis. The other extreme in the design space would be an entirely light-weight mechanism that operates only on the coarse-granularity data. Both these extremes have shortcomings. The light-weight mechanism incurs very little computational cost, but lacks the desired investigative capabilities that more fine-grained analysis can provide. The heavy-weight mechanism, on the other hand, incurs a much higher collection and computation cost. Further, the heavy-weight mechanism may be operating in an *un-focused* manner, i.e., without knowledge about the seriousness of the incidents that actually need operators' attention. Operating in such a setting is detrimental to not only the scalability (due to high collection and computation complexity), but also the accuracy (the false alarm rate may be high).

Our work attempts to find an operationally convenient space between these extremes. The key idea in our approach is to use *possible* anomalous events detected in coarse grained data *close* to the attack target, to *focus* the search for anomalies in more detailed data. A simple but powerful observation, is that close to the attack target, e.g., at the customer access link, detecting flooding attacks using coarse-grained data becomes reasonably easy. Even though such coarse-grained indicators might generate false alarms and lack the ability to generate useful attack reports, they can be used to guide further, more fine-grained, analysis. Based on this insight, we propose a triggered multistage detection approach (Figure 2) in which the successive stages can have access to and operate on data streams of increasing granularity. A triggered approach helps focus collection and computation resources intelligently – performing inexpensive operations early on, but allowing sophisticated, data and compute intensive tasks in later stages.

3.2 Design Alternatives

While our approach generalizes to any number of detection stages, the system described in this paper is limited to two stages. We briefly describe our methods as well as other potential alternatives for each stage.

3.2.1 Lightweight Anomaly Detection

As the name suggests the key constraint here is that the method not require significant processing so that it can be applied to a large number of interfaces. Since the output of this stage triggers more detailed analysis in the second stage, false positives are less of a concern than false negatives. A false positive from the first stage will only cause more unnecessary work to be done in the second stage, and hence not necessarily much of a concern for the operator who only sees the final alarms after the second stage. However, a false negative is a more serious concern as the attack might be missed altogether.

Volume anomaly detection: Traffic anomalies on volume and link utilization data available from egress interfaces are often good indicators of flooding attacks. Traffic metrics that are available from most current router implementations include the traffic volume (either in bytes per second or packets per second), router CPU utilization, and packet drop counts. Our approach (described in Section 4.1) involves the use of traffic time-series modeling to predict the expected future load on each customer-facing interface. These prediction models are then used to detect significant deviations to identify future volume anomalies.

Using traffic distribution anomalies: Lakhina et al. [20] discuss the use of distributions (using entropy)

for diagnosing anomalies in networks. The key insight behind their work is that many attacks can be identified by substantial changes in traffic distributions, specifically the distribution across source and destination addresses and ports. While the use of distribution information in [20] was suggested as a means for augmenting volume-based anomaly detection, we can use distribution anomalies as triggers for further analysis. The use of such metrics in a triggered approach may not necessarily reduce the collection cost, since measuring these metrics may need access to very fine-grained traffic information.

3.2.2 Focused Anomaly Detection

Even though our triggered approach will reduce the search space for the second stage significantly, the scale of the problem is such that the computational overhead remains a concern. The other key requirements are accuracy, and the ability to generate useful incident reports for network operators.

Rule-based detection: Some DDoS attacks have distinct characteristics that can be easily captured with a small set of detection rules. For example, a large number of single-packet flows (e.g., ICMP-ECHO or TCP-SYN) sent to a single destination IP address is often indicative of a DDoS attack. Another option is to use *botnet blacklists* to check if the set of source addresses that occur frequently in the traffic belong to known compromised machines (commonly referred to as zombies) used for launching attacks. Rule-based approaches have near-term appeal since they typically have low false-positive rates, even though their detection capabilities are limited to the set of attacks spanned by the rule-sets.

Uni-dimensional aggregation: Our specific implementation for the second stage involves the use of uni-dimensional hierarchical aggregation algorithms. Conceptually, uni-dimensional clustering attempts to discover heavy-hitters along source/destination prefixes, using a thresholding scheme to compress reports along the prefix hierarchy. Since the computational overhead with performing uni-dimensional aggregation is low, and the functionality provided is sufficient for investigating most known types of DDoS attacks we choose this approach. Our implementation, which is a combination of uni-dimensional clustering and rule-based approaches, is described in Section 4.2.

Multi-dimensional clustering: Multi-dimensional clustering provides another alternative for fine-grained analysis [11, 32]. The basic theme of these approaches is to abstract the standard IP 5-tuple (srcaddr, dstaddr, protocol, srcport, dstport) within multi-dimensional clustering techniques to report traffic patterns of interest. Typically, the complexity of the algorithm can be reduced by tuning the technique to report only interesting clusters, those

that either have a high volume or those that have a significant deviation from an expected norm.¹

3.3 Benefits and Pitfalls

The benefits of our triggered approach are as follows:

Detecting high-impact attacks: Since our triggers are generated close to the customer egresses, we are more likely to detect attacks that actually impact the end-user. Note that this is in contrast to more centralized approaches, even those that work on more fine-grained data feeds.² For example, by monitoring SNMP byte counts on a T1 access link it is straightforward to determine when the link is being overloaded. Looking for the same information from a different vantage point, e.g., at a set of major peering links is a much more challenging task. Not only could the traffic flowing towards the T1 interface be spread across many such peering interfaces, but the traffic will be hidden inside an overwhelming amount of background traffic on the peering links.

Efficient data collection: SNMP data is lightweight enough that it can be collected on the access routers without imposing significant load. Netflow data, on the other hand, can be more efficiently collected at more powerful and better-provisioned core routers so that access routers are not burdened with this more expensive process.

Reduced computation cost: We use high cost operations and expensive algorithms in a focused manner, and also significantly reduce the data volumes that the expensive operations need to handle.

Low operational complexity: The different stages are simple and easy to understand, and vendor-independent, and managing the operation should be relatively simple. More importantly, our implementation works with data streams that are already available to most provider networks. Deploying LADS does not incur any overhead in terms of instrumenting new monitoring capabilities or deploying special hardware for data collection and analysis.

Near real-time incident reports: Since the computational complexity is significantly reduced, we can operate the system in near real-time, without relying on specialized hardware or data structure support.

Flexibility: Our approach is flexible in two aspects. First we can easily accommodate other data streams as and when they are available. Second, within each stage the performance and algorithms can be optimized to reach desired levels. For example, our first stage trigger currently uses simple time-series volume anomaly detection. It is fairly easy to augment this step with other data streams and traffic metrics, or alternatively use other anomaly detection methods for the same dataset.

In our approach, there are four potential pitfalls. The first pitfall is one relating to possible undesirable interac-

tions between the trigger stage and the detailed analysis stage. While our approach allows for each component to be optimized in isolation, optimizing the overall system performance would require a detailed understanding of the interfaces and interactions between different components. Managing and optimizing such multi-component systems is inherently complicated – we believe our specific implementation is based on a clean set of interfaces between components which are sufficiently decoupled, and hence has very few, if any, undesirable interactions.

The second, more serious problem, is one of misses due to the triggered approach. While the low-level triggers reduce the collection and computation complexity, they may be doing so by compromising the sensitivity of the system, i.e., by increasing the false negative rate. Attacks which can cause the greatest disruption in terms of traffic engineering, routing etc., are flooding attacks, and these will invariably show up as volume anomalies on the egress interfaces closest to the customers. Since our primary focus is on such flooding attacks, there is almost no impact on the false negative rate. The benefits we gain in terms of operational simplicity and reduced false alarm rate greatly outweigh the negligible decrease in the detection sensitivity toward low-impact attacks.

The third pitfall is related to the ability of the monitoring infrastructure to sustain data collection during attacks. While collecting low-volume SNMP feeds is not a serious overhead, collecting flow records at the customer egresses and transporting them back to a centralized processing engine is clearly infeasible during volume floods. The access link is already overloaded, and reporting large volumes of flow records can only worsen the congestion. Large providers typically deploy flow collectors at core network elements, which are usually well-provisioned, and they can subsequently map the flow records to the appropriate egresses using routing and address space information. Thus, there will be no perceivable reduction in the data collection capabilities during attacks.

Finally, there is a concern regarding the resolution limits of in-network DDoS detection – whether such an approach can detect anomalies on all possible scales of network prefixes. Our system deals primarily with flooding attacks that impact immediate customers directly connected to the provider network. Our experiences with both the SNMP and the flow analysis indicates that at this granularity, LADS is effective at identifying anomalies and providing sufficient information for operators to respond to the alarms.

4 Implementation

Our implementation of LADS, currently works as an off-line DDoS detection system within a tier-1 ISP. The described implementation works on real traffic feeds of the

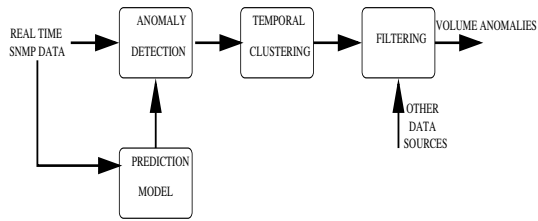


Figure 3: Overview of SNMP Anomaly Detection

tier-1 ISP and is only classified as off-line in that the data currently provided to the system might be substantially delayed. We are actively working on deploying the system in an on-line environment in which real-time data feeds are available. Our performance evaluation (Section 6.1) indicates that our design and implementation will be adept to the task of on-line monitoring.

4.1 Lightweight Anomaly Detection

The first stage of the detection system uses SNMP link utilization data to report volume anomalies. Figure 3 provides a conceptual overview of the SNMP anomaly detection module. Specifically, we are interested in flow anomalies on customer-facing interfaces, since volume based DDoS attacks will be most visible at the egress links of the customers under attack. SNMP data is collected on an on-going basis at most providers, and typically contains CPU and link loads (in terms of octet and packet) counts. Since most DDoS attacks use small packets we use the egress packet counts (i.e., with reference to Figure 1, the C-PE interface towards the customer) to detect volume anomalies.

To keep the operational, storage, and computation resources low, we devised a simple trigger algorithm with good performance. Conceptually, the algorithm builds a prediction model which indicates the expected mean and expected variance for the traffic time series. Using this model it assigns a deviation score to current observations, in terms of the number of standard deviations away from the mean that each observation is found to be. Borrowing some formal notation [26] one can think of the traffic time series, denoted by $T(t)$ as being composed of three components, $T(t) = P(t) + V(t) + A(t)$, where $P(t)$ represents the predicted mean traffic rate, $V(t)$ represents the stochastic noise that one expects for the traffic, and $A(t)$ is the anomaly component. Our goal is to obtain models for P (the periodic component) and V (the stochastic component), so that we can identify the anomaly component A in future traffic measurements.

Our algorithm, depicted in Figure 4, works as follows. For each customer interface, we take the last k weeks of data (this historical time-series data is referred to as TS). We build an empirical mean-variance model by simple point-wise averaging, assuming a basic period-

TIMEDOMAINMODELING(TS, W, N)

```

// TS is the training set
// W is the number of weeks
// N is the number of data points per week
1 for  $i \leftarrow 1$  To  $N$  do
    //  $P$  is the periodic mean
2    $P(i) \leftarrow \text{MEAN}(TS(1 : W, i))$ 
    // Fourier denoising
3    $P' \leftarrow \text{DENOISE}(P)$ 
    //  $V$  is the variance
4    $V(i) \leftarrow \text{VARIANCEMODEL}(TS, P', W, N)$ 
5 return  $P', V$ 

```

Figure 4: Procedure for timeseries modeling

ANOMALYDETECTION(T, TS, W, N)

```

// T is the new time series
// TS is the historical time series
// W is the number of weeks for building model
// N is the number of data points per week
1 ( $P, V$ )  $\leftarrow$  TIMEDOMAINMODELING( $TS, W, N$ )
2 for  $i \leftarrow 1$  To  $N$  do
3    $D(i) \leftarrow (T(i) - P(i))/V(i)$ 
4   TEMPORALCLUSTER( $D, \alpha_{\text{trigger}}, \alpha_{\text{add}}, \text{keepalive}$ )
5 Use filtering rules on clustered alarms

```

Figure 5: Outline of SNMP anomaly detection

icity of one week. For example, for estimating the expected traffic for the 5 minute interval *Fri 9:00-9:05 am*, we take the average of the observations over the past k Fridays, for the same 5 minute timeslot. As the training data might itself contain anomalies we perform a denoising step using a Fourier transform from which we pick the top 50 energy coefficients.³ In the final step, for each point per week (e.g., *Fri 9:00-9:05, Mon 21:00-21:05*), the algorithm determines the variance over the last k observed data points with respect to the de-noised historical mean. The implicit assumptions in the method are that the basic periodicity of the traffic data is one week and that traffic patterns are relatively stable week to week, which has been suggested in other traffic analysis on similar datasets [1, 26].

Figure 5 outlines the four main stages in the anomaly detection process. We first obtain the historical prediction model, to get the deviation scores. Then, we use the estimated model to obtain deviation scores for current SNMP data to obtain volume anomalies. We use a natural definition of the deviation, $D(t) = (T(t) - P(t))/V(t)$, which represents the number of standard deviations away from the prediction that the observed data point is. Once the deviation scores have been computed, we perform a temporal clustering procedure (Figure 6) to report anomalous incidents to the flow collector and analyzer. Temporal clustering can reduce the load

```

TEMPORALCLUSTER( $D, \alpha_{trigger}, \alpha_{add}, keepalive$ )
    //  $D$  is the deviation score time series
    //  $\alpha_{trigger}$  is trigger deviation score threshold
    //  $\alpha_{add}$  is the event extension threshold
    //  $keepalive$  is the time for which an event is active
1  if there is a current active event  $E$ 
    then
2      if  $D(currenttime) \geq \alpha_{add}$ 
        then
3          Extend the current event  $E$ 
4          if  $((Starttime(E) - currenttime) > keepalive)$ 
            then
5              Expire the current event  $E$ 
        else
            // Check if new event has occurred
6          if  $D(currenttime) \geq \alpha_{trigger}$ 
            then
7              Create a new anomaly event
8  Return anomaly events, with start and stop times

```

Figure 6: Temporal clustering to reduce query load

on the collection mechanism by reducing the number of queries that we issue to the collector. Such a load reduction is indeed significant, as many attacks do last quite long. The clustering method operates based on two pre-defined deviation score thresholds, the event trigger threshold ($\alpha_{trigger}$) and the event extension threshold (α_{add}), as well as a keep alive time ($keepalive$). The clustering process tries to extend the current active event, if the new observation has a deviation score that exceeds the event extension threshold α_{add} , within a time duration of $keepalive$, since the start of the event. If there is no active ongoing event, it creates a new event if the observed deviation score is higher than the event trigger threshold ($\alpha_{trigger}$).

After detecting the SNMP anomalies, we perform additional filtering steps to allow the operators to remove known or uninteresting anomalies. We use an absolute volume threshold to remove all SNMP alarms which have an average bandwidth less than a pre-defined threshold. This allows the operator to specify a minimum attack rate of interest, to reduce the overall workload for the flow collectors. Second, we remove anomalies in the SNMP data caused by router resets and SNMP implementation bugs. In particular we remove the first SNMP counters after a reset (e.g., we saw in a few cases, immediately after a reset, a SNMP counter corresponding to -1), as well as measurements which indicate a bandwidth utilization greater than the physical bandwidth. Even though such events are extremely rare, they do occur daily on a large network, and we remove such measurement anomalies.

At the end of the SNMP anomaly stage, we receive a set of volume anomalies, each anomaly being specified

by the egress interface, and the start and end time. This information is then used to trigger Netflow collection for detailed investigation.

4.2 Focused Anomaly Detection

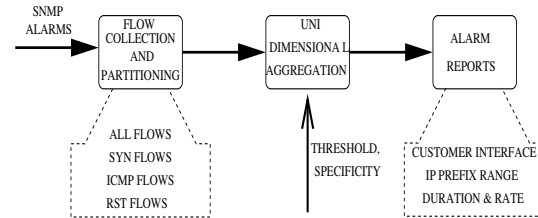


Figure 7: Incident diagnosis using flow data

The second stage of our DDoS detection system performs detailed analysis on Netflow data as shown in Figure 7. We first collect all Netflow records for the egress interface indicated by the first stage trigger. Section 5.1 describes the collection infrastructure in greater detail. Then for each SNMP-alarm we build the following Netflow datasets containing,

- Records with the TCP SYN flag set (*SYN* set)
- Records with the TCP RST flag set (*RST* set)
- Records for ICMP flows (*ICMP* set)
- All flow records (*All* set)

Finally, for each of the Netflow datasets, we report the traffic volumes for all destination prefixes with a prefix length larger than a /28, using the uni-dimensional clustering algorithm described in Figure 8. The algorithm generates a bandwidth attack alarm if the *All* set contains a prefix smaller than /28 which carries more traffic than the configurable *Bandwidth Attack Threshold*. It will also generate a SYN/ICMP/RST alarm if the corresponding SYN/ICMP/RST sets observe an IP prefix range which carries more traffic than the configurable SYN/ICMP/RST Threshold. Instead of using a fixed rate threshold, we use a duration-adaptive rate threshold mechanism, which takes into account the duration of the SNMP volume anomaly. This will balance the sensitivity between high-rate low duration attacks, and relatively lower-rate but longer duration attacks. This can be achieved by using a simple rate-depreciation approach, so that the rate threshold is a monotonically decreasing function of the alarm duration. Our current implementation uses a geometrically decreasing depreciation, where the average rate for longer duration events will be generated according to the following formula $Rate(Duration) = Rate(BaseDuration) * DecreaseFactor^{\frac{Duration}{BaseDuration}}$, where the *BaseDuration* is 300 seconds, and the *DecreaseFactor* is set to 0.95.

There are two steps of the uni-dimensional clustering (Figure 8): Aggregation and Reporting. The aggregation

step simply counts the total traffic volume received by each distinct destination prefix, larger than a minimum prefix-range size, denoted by *MinPrefix*. Since we are interested in DDoS attacks on customer egress links, we can afford to perform the traffic aggregation on smaller prefix ranges, than would be the case for more general purpose traffic analysis applications [11, 32, 35]. Thus the computational and memory overhead during the aggregation phase is upper-bounded by the prefix range we are interested in. For example, we are only interested in the traffic intended for prefixes smaller than a /28, which can be potential attack targets. The next step is the *Reporting* stage, which uses the aggregated counters to decide whether to report each particular prefix range as a potential attack target. The reporting step is conceptually similar to the work of Estan et al. [11] and Singh et al. [35], to generate traffic summaries indicating heavy-hitters. Intuitively, we generate reports on larger prefixes, if they carry substantially more traffic than a previously reported smaller prefix range, and if they are above the absolute volume threshold. We scale the absolute volume threshold according to the size of the prefix range by a multiplicative *Specificity* parameter that determines the scaling factor. We chose this approach due to its simplicity and we observe that the diagnostic capability provided by our approach is sufficient for detecting DDoS attacks, and generating alarm reports comparable to commercial DDoS solutions (Section 6.4). We found in our experiments that this approach is computationally efficient, in terms of memory and processing time, which makes it a viable alternative for real-time analysis.

5 Experimental Setup

To evaluate our LADS implementation we collected SNMP and Netflow data for a subset of the access interfaces of a tier-1 ISP ranging from T1 to OC-48 speeds. We describe this data collection next followed by a description of the LADS parameter settings we used.

5.1 Data Description

For our LADS evaluation we collected SNMP and Netflow data for over 22000 interfaces within a large tier-1 ISP. To allow our evaluation to be repeatable during development we archived all relevant data for an eleven day period in August 2005 with the exception of the SNMP data used which was archived for a period in excess of 12 months. We also collected alarms from the commercial DDoS detection system for this period.

SNMP Feeds The SNMP reports are generated from each egress router within the tier-1 ISP and reported periodically for each 5 minute interval. For each interface,

the SNMP reports contain the total traffic volume per interface (both packet and byte counts), and router utilization information for the recent 5 minute interval. The reporting interval can be configured in most router implementations. Within the ISP, this value is set to 5 minutes – small enough to initiate real-time response, but large enough to keep the router overhead low. The total collection overhead for SNMP data over the entire provider's network is around 200 MB of compressed (ASCII) data per day, which represents a small bandwidth overhead compared to large volumes (of the order of few petabytes per day) of traffic that a tier-1 ISP carries. In LADS, we only use the egress packet counts, i.e., the packets from the ISP toward the customer, to generate triggers.

Netflow Feeds The Netflow collection infrastructure collects sampled flow records covering the entire backbone network (more than 500 routers within the ISP). The records are based on 1:500 packet sampled data. The sampling is performed on the router and the records are subsequently smart sampled [10] to reduce the volume. In smart sampling, flow records representing a total volume greater than a threshold of 20 MB are always sampled, while smaller records are sampled with a probability proportional to their size. Appropriate renormalization of the reported volume (in bytes) yields unbiased estimates of the traffic volume prior to sampling [10]. In the resulting data set each record represents, on average, at least 20 MB of data. After collecting the records we annotate each record with its customer egress interface (if it was not collected on the egress router) using route simulation and tag records which could have been observed twice within the collection infrastructure to avoid double counting of flow records. We emulate a triggered flow retrieval system on the entire set of smart sampled flow records. i.e., we query the flow data available from all collectors to obtain the flow records relevant to each SNMP anomaly. Since our current implementation runs in off-line emulation mode, the benefits of a triggered collection approach are not realized.

Alarms from commercial system The ISP has a commercial DDoS detection system deployed at key locations within its network. We collected the high priority DDoS alarms from this commercial DDoS detection system. The alarms were combined into attack records if we found multiple alarms for the same target with an idle time of less than 15 minutes in between alarms. Even though we are not aware of the detailed algorithms used within this product, operational experience indicates that the system detects most large DDoS attacks while generating a manageable number of high priority alarms. The system is deployed in a substantial fraction of the core of the ISP at high speed interfaces and, therefore, only analyzes aggregate customer traffic. We use the commercial detection system as a basis for comparison with our im-

```

UNIDIMENSIONALCLUSTERING(MinPrefix, Threshold, Specificity)
    // MinPrefix is the minimum prefix length – Set to 28, MaxPrefix is the maximum IP prefix length (32 for IPv4)
    // Threshold is given in terms of an attack rate, Specificity is used for compressing the report – Set to 1.5
1  Aggregation: Read flow records and update traffic counts for prefixes between MinPrefix and MaxPrefix
2  Reporting: for  $i \leftarrow \text{MaxPrefix}$  DownTo MinPrefix do
3      for each prefix  $P$  of prefix-length  $i$  do
        // We use the IP/prefix notation,  $P/\{i\}$  refers to prefix  $P$  with a prefix-mask of length  $i$ 
4       $\text{AbsoluteThreshold} \leftarrow \text{Specificity}^{(\text{MaxPrefix}-i)} \times \text{Threshold}$ 
5      if  $i \neq \text{MaxPrefix}$ 
        then
6           $\text{CompressThreshold} \leftarrow \text{Specificity} \times \text{PredictedVol}(P/\{i\})$ 
        else
7           $\text{CompressThreshold} \leftarrow 0$ 
8       $\text{ReportThreshold} \leftarrow \text{MAX}(\text{AbsoluteThreshold}, \text{CompressThreshold})$ 
9      if  $\text{Volume}(P) > \text{ReportThreshold}$ 
        then
10         Report alarm on prefix  $P/\{i\}$  with rate  $\text{Volume}(P)$ 
11          $\text{PredictedVol}(P/\{i-1\}) \leftarrow \text{MAX}(\text{PredictedVol}(P/\{i-1\}), \text{Volume}(P))$ 

```

Figure 8: Procedure for uni-dimensional prefix aggregation and generating compressed reports

plementation even though we are aware that due to its deployment locations and configuration (we only collect high priority alarms) the commercial system might not detect some of the DDoS attacks which are detectable with our system. In an ideal scenario, we would like to evaluate the false positive and false negative rates of our system against some *absolute ground truth*. We are not aware of any system which can generate such ground truth at the scale that we are interested in, and this commercial system is our closest available approximation despite its inherent limitations.

5.2 System Configuration

In terms of the specifics of our implementation, our approach requires a number of configurable parameters which we set to the following values:

SNMP training period The training period for model building for SNMP anomaly detection is $k = 5$ weeks.

Absolute Volume Threshold The absolute volume threshold provides a lower bound on DDoS attacks we detect in the SNMP data. We set this value to 250 Kbps which considering that the smallest link size in the Tier-1 ISP's network is a T1 (1.5 Mbps) allows us to detect any sizable attack on any interface under consideration.

Event Score Threshold ($\alpha_{trigger}$) The threshold on the deviation score which triggers an SNMP-based alarm. We evaluate the sensitivity and overhead for different threshold values in Section 6.2.2. For our evaluation we use $\alpha_{trigger} = 5$.

Temporal Clustering Parameters The temporal clustering procedure uses an event extension threshold (α_{add}) and a *keepalive* duration value, for deciding on

combining SNMP anomalies. We set $\alpha_{add} = 2.5$, and the *keepalive* duration to be 15 minutes.

Bandwidth Attack Threshold This threshold is applied to determine if a particular incident should be reported as a potential DDoS attack, if none of the other DDoS related signatures (e.g., high volumes of SYN, ICMP, or RST packets) are present. We set this threshold to a high-intensity threshold of 26 Mbps,⁴ targeted at a single /32 behind a customer interface. The rate for alarms of longer duration will be lower due to the rate depreciation described in Section 4.2. The thresholds for larger prefixes (upto /28) are scaled according to the algorithm described in Figure 8.

SYN/ICMP/RST Threshold This threshold is applied to determine within the flow data if a particular incident could be considered a SYN, ICMP or RST attack. Currently we set this rate to a high intensity rate of 2.6 Mbps,⁵ averaged over a 300 second interval. Again, we use a similar rate depreciation function for longer duration alarms.

6 Experimental Results

We first study our system performance in Section 6.1, followed by an evaluation of the SNMP based trigger phase in Section 6.2, before analyzing the incidents generated by our system in Section 6.3 and Section 6.4.

6.1 Performance

The data was collected using an existing SNMP and Net-flow data collection infrastructure. The SNMP data is being collected by a commercial off the shelf SNMP

collection tool which seems to scale easily to large networks. The Netflow collection system on the other hand was specifically build for this large ISP and is described in more detail in [10]. Currently this infrastructure monitors in excess of one petabyte of traffic each day.

Using these existing data sources we implemented our data extraction using a combination of flat files and an in-house database system. The data-extraction and analysis modules were implemented in Perl. The model-building phase uses additional MATLAB scripts for performing the de-noising operation.

The model-building phase uses 5 weeks of data per interface to get a mean-variance model for the anomaly detection. It took roughly 26 hours to perform the data extraction, de-noising, and model extraction for all the 22000 interfaces. This is not a concern since this part of the analysis can be performed offline, as it is not on the critical path for real-time attack detection.

We ran LADS in off-line emulation mode for our entire 11 day period on a heavily shared multi-processor 900MHz SUN Ultra. The anomaly detection stage was parallelized using 6 processes, and it took 11.2 seconds to report volume anomalies (i.e., finding deviation scores, doing the clustering, and filtering out measurement errors), for each 5 minute interval across the 22000 interfaces. The biggest bottleneck for our performance is the extraction of flow records for each reported SNMP volume anomaly (even after the reduction due to the triggers). The main reasons being (a) all flow data is compressed to meet storage constraints, and (b) the flow data is collected and indexed on a per-collector basis and not indexed based on the egress interface. Even with these performance inhibitors, it takes around 212.5 seconds to extract the flow data that needs to be analyzed. We note that this time can be reduced significantly by indexing the data appropriately.

The last stage of our analysis performs uni-dimensional aggregation on the collected flow data, taking approximately 40 seconds for each 5 minute interval. Thus, for each 5 minute interval of data arriving at the processing engine, the total time that is needed to report alarms, is the sum of the time taken to generate SNMP anomalies, the time taken to extract flow data for the triggered data, and the time taken to process the flow data, which is equal to $11.2 + 212.5 + 40 = 263.7$ seconds. The resulting maximum latency with which we will report an alarm is, therefore, at most 263.7 seconds, implying that even with our current implementation (with very few performance optimizations) we can perform near real-time attack detection. On a more state of the art platform (900MHz UltraSparcs are quite dated!), with additional implementation optimizations and better data indexing we can achieve substantially better performance.

6.2 SNMP-based Trigger Evaluation

We evaluate our SNMP based trigger implementation in three stages. First, we discuss the choice of our trigger algorithm, then we compare our trigger events against the commercial-alarms and finally we highlight the savings our triggered approach provides.

6.2.1 Choice of Algorithm

In the context of our system we are looking for a model which is efficient, uses only historically available data and detects anomalies early. Those requirements are motivated by the fact that we have to perform this analysis in real time on tens of thousand of times series to provide DDoS alarms within a reasonable timeframe. Our mean-variance based model provides these features.

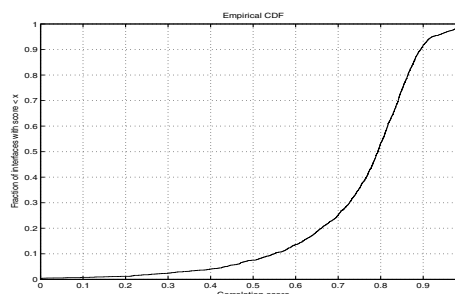


Figure 9: Correlation with the anomaly detection procedure proposed by Roughan et al. [26]

As a preliminary verification of the correctness of our anomaly detection system, Figure 9 depicts the correlation of our trigger algorithm with the one proposed by Roughan et al. [26]. The basic difference between these approaches lies in the assumption about the variance of the time-series. We use an empirical data-driven approach while Roughan et al. [26] assume that the stochastic component is of the form $\sqrt{a} \times P_t$, where a is a peakedness factor, and P_t is the periodic component of the time-series model (obtained using moving averages). Figure 9 shows a correlation score of greater than 0.7 between these two methods for more than 75% of all the 22000 interfaces selected for detection.

We find that in our problem domain, the simple trigger model has similar properties to more complex models and is adequate to perform the trigger function. One direction for future work is to compare our lightweight detection mechanism with other algorithms for time-series anomaly detection [34].

6.2.2 Accuracy

The objective of the trigger phase of our system is to reduce the flow data collected and analyzed to a manageable amount, and not to diagnose attacks directly. Therefore, an interesting question is the sensitivity of the trig-

gers with respect to known actual attacks – how often and by what magnitude do the known flooding attacks show up as volume anomalies.

To evaluate the sensitivity of the trigger, we use synthetically introduced volume anomalies of varying magnitude, and tested the detection false negative rate as a function of the detection threshold. For each of the 22000 interfaces under analysis, we introduce 20 randomly located anomalies for a chosen magnitude. Figure 10 shows the false negative rate for different thresholds ($\alpha_{trigger}$). We notice that the false negative rates are indeed low, and that with anomalies of increased magnitude the false negative rates drop off quite significantly. Also, we notice that for larger anomalies, the false negative rate is expectedly less sensitive to the choice of the anomaly detection threshold.

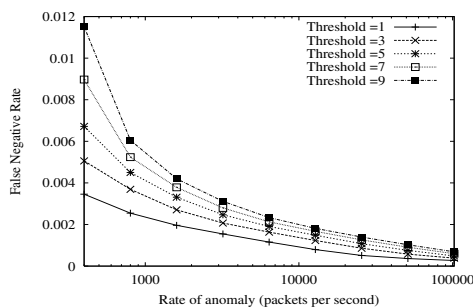


Figure 10: Evaluating false negative rate with synthetic volume anomalies

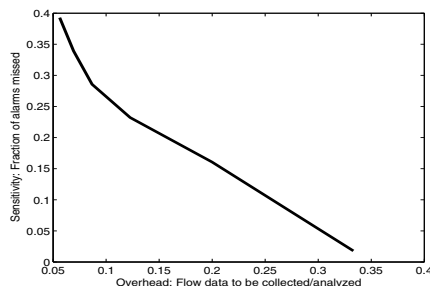


Figure 11: Tradeoff between sensitivity and scalability

A related question is how frequently the SNMP based triggers miss an attack in the commercial-alarm set. Figure 11 depicts the tradeoff between the sensitivity of the triggers and the overall data complexity reduction the trigger can achieve. The sensitivity of the anomaly detection module, for a particular deviation score threshold, is the percentage of commercial-alarms which match a SNMP anomaly for the threshold. The data complexity reduction achieved by the trigger can be calculated in terms of the flow data that will be collected after the trigger step and which needs to be further analyzed. Ideally, we would like to have a trigger that has perfect sensitivity (i.e., zero false negative rate), that can provide very low collection overhead for later stages.

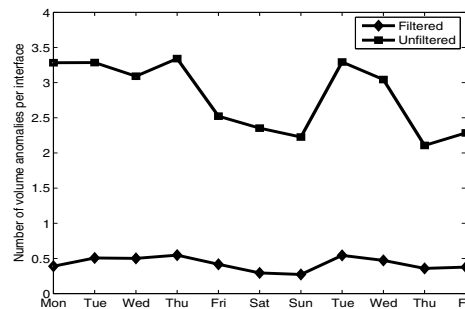


Figure 12: Number of SNMP events per interface per day

As a tradeoff between sensitivity and data reduction we chose an anomaly detection threshold of 5. This results in an 85% overlap with the commercial-alarms, and corresponds to an 80% data reduction (i.e., only 20% of the original flow data needs to be collected and analyzed in the second stage of our system). The reduction in collection overhead is significant considering that the total bandwidth overhead for collecting (1 in 500 sampled) flow records is around 2-3 TB per day, for a tier-1 ISP. With an 80% reduction rate, the total bandwidth overhead for collecting flow records would be roughly 40 Mbps over the entire network which is quite manageable. From a provider's perspective, the alarms that are of paramount importance are those that affect the customers the most, and typically these are attacks which overload the egress link. If the misses occur on well-provisioned interfaces, the loss in sensitivity is not a serious concern. We will further discuss the alarms that do not appear as volume anomalies in Section 6.4.

Figure 12 shows the number of SNMP anomaly events per customer interface per day over the evaluation period before and after applying the filters. The filtering reduces the number of SNMP anomaly events on average by a factor of 6 (the predominant contribution being one from the absolute volume threshold). Considering the fact that these events will be automatically processed by the second phase of LADS, the number of SNMP anomalies is quite manageably low.

6.3 Incident Analysis

Next we characterize the alarms generated by LADS after performing the Netflow analysis described in Section 4.2. Each alarm specifies a duration and a customer-facing interface. It also contains a set of destination IP-prefixes which are receiving high incoming traffic volumes (and hence potential DDoS targets), the bandwidth of the suspected DDoS event, along with the alarm-type (BW, SYN, RST, ICMP).

Figure 13 shows the number of alarms during our 11-day evaluation period.⁶ Here we consider incidents at the

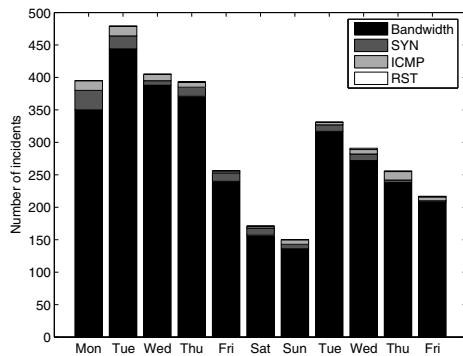


Figure 13: Number of reported incidents (at egress interface granularity) in 11 day period in Aug 2005

granularity of egress interfaces, i.e., concurrent floods against multiple IP addresses on the same egress interface will be considered a single incident. We generate approximately 15 incidents per hour which seems reasonable considering that we monitor 22000 customer interfaces and that this number of incidents could easily be handled by the security staff of a large network provider. We observe that a large fraction of the incidents are reported as potential bandwidth attacks.

The number of distinct IP addresses involved in an attack might be a better indicator of the workload for operators to investigate these alarms. If we split all alarms which target multiple IP addresses at a single egress interface into multiple alarms, we only see a slightly higher alarm rate (around 17 per hour). That is not surprising considering that most (76%) of the incidents involve only one IP address. We also find that in around 19% of the cases we get repeated alarms for the same IP address within the same day. These alarms would most likely only require one investigation by the network operator. Therefore, we believe that the above alarm rates are actually an upper bound of the number of trouble tickets network operators need to process.

Some of these alarms may indeed be benign bandwidth floods or flash crowds. Nevertheless, they are a cause of concern and we believe there is value in bringing these incidents to the attention of network operators. First, in the absence of ground truth regarding these incidents, such incidents should be brought to the notice of network operators. Second, since there was a large volume anomaly on the interface, with one or more destination IPs receiving a high data rate during this time, such reports may benefit other aspects of network operations such as traffic engineering and route management. We are investigating other techniques to independently characterize these alarms (e.g., [14]).

6.4 Comparison with Commercial DDoS Detection System

In this section we compare LADS alarms against the commercial-alarms. Since the commercial DDoS detection system only covers key locations and we only receive high level alarms from this system we would expect that our alarm set contains substantially more attacks than the commercial-alarms. The objective of LADS is not necessarily to provide better detection than the commercial system on the links covered by the commercial system. Rather the goal is to provide detection functionality comparable to deploying the commercial system on all the interfaces of the ISP, but to do so at a fraction of the cost that would be incurred in having to deploy the commercial system on every interface. Hence, we use this data set primarily to evaluate the false negative rate of LADS on the links covered by the commercial system.

Figure 14 presents a breakdown of the comparison of LADS alarms versus the commercial-alarms. The breakdown uses the following categories to classify the 86 alarms obtained from the commercial system.

Successes Between the LADS alarms and the commercial-alarms the interface matches, the IP prefix alarmed matches, and the durations of the reported alarms overlap.

Found early incidents Between the LADS alarms and the commercial-alarms the interface matches, the IP address alarmed matches, but we find the alarm slightly earlier than what is reported.

Found late incidents Between the LADS alarms and the commercial-alarms the interface matches, the IP address alarmed matches, but we find the alarm slightly later than what is reported by the commercial system.

Anomaly detection misses There is no SNMP event on the interface corresponding to a commercial-system alarm, i.e., the deviation score for the corresponding time is less than the threshold ($\alpha_{trigger} = 5$).

Potential commercial-alarm false positive The interface information and the anomaly match between our SNMP alarm and the commercial-alarm. However, we find little or no flow data for the corresponding attack target reported by the alarms.

Threshold misses We have an SNMP volume anomaly, and we have flow data for the commercial-system alarm. We find quite a large number of flows to the IP, but LADS did not raise an alarm for the IP address.

The false negative rate of our system compared to the commercial DDoS detection system is essentially the sum of the anomaly misses, and threshold misses. Manual analysis of the anomaly detection misses indicates that all 7 SNMP anomaly misses are caused by relatively small attacks on OC-48 interfaces (2.48 Gbps). They did not saturate the customer interface and therefore are not

in the category of DDoS attacks of immediate concern. The number of threshold misses on our system is low - just one out of 86 incidents are missed due to the threshold settings. We conclude that the overall false negative rate of our system, compared to a commercial DDoS detection system, is 1 out of 80, or 1.25%.

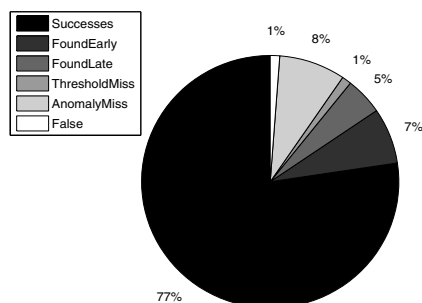


Figure 14: Comparison with the proprietary system

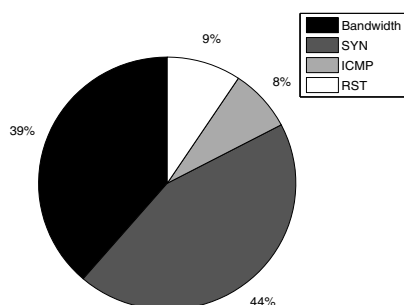


Figure 15: Breakdown of overlapping incidents

Next we proceed to analyze the incidents that are common to both LADS alarms and the commercial-system alarms. We give a breakdown of the 4 types of incidents *Bandwidth*, *SYN*, *ICMP*, *RST* in Figure 15. Interestingly, the largest portion of the reported incidents which overlap are SYN floods. Figure 16 shows the average bitrate (calculated from the flow records) of the DDoS alarms generated by our system and the commercial DDoS detection system. The overlapping incidents appear to have a minimum rate of 10 Mbps, which is most likely due to the fact that we only had access to the high priority alarms of the commercial DDoS detection system. Interestingly, this makes the high level alerts of this system unsuitable for detecting DDoS attacks against small customer links. Since, it is primarily deployed in the core the system ranks attacks as high level alerts not by customer impact but by the overall attack size. This is of course less desirable, if the goal is to protect customers with diversity in subscription line rates. For 40% of the LADS alarms we find a reported bandwidth which is smaller than 10Mbps.⁷ Further investigation reveals that more than 70% of these low volume alarms are in fact caused by volume floods against low speed links.

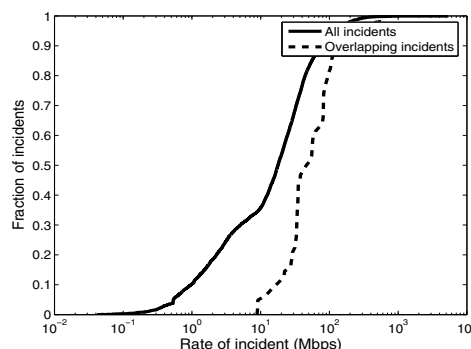


Figure 16: Rates of potential attack incidents

7 Conclusions

We presented the design of a triggered framework for scalable threat analysis, and a specific implementation based on SNMP and Netflow feeds derived from a tier-1 ISP. Our evaluations and experience with large networking datasets demonstrate the need for such an approach. LADS, our large-scale automated DDoS detection system, can provide detection and diagnostic capabilities across all customer interfaces of a large tier-1 ISP, without relying on the deployment of additional hardware monitoring devices. Of particular practical significance is the fact that our system uses data feeds that are readily available to most providers. We are investigating other ways in which we can confirm the validity of the alarms generated by our system (for example, using customer incident reports and other historical traffic profiles). Finally, we are currently pursuing the implementation of real time data feeds to our system to allow us to use it as an online attack detection mechanism.

Acknowledgments

We wish to thank Jennifer Yates and Zihui Ge for helping us with collecting the SNMP data. We acknowledge the valuable feedback provided by Albert Greenberg. We thank Vidhyashankar Venkataraman and the anonymous reviewers for their comments on the paper, and thank Geoff Voelker for shepherding our paper.

References

- [1] AIELLO, W., GILBERT, A., REXROAD, B., AND SEKAR, V. Sparse Approximations for High-Fidelity Compression of Network Traffic Data. In *Proceedings of ACM/USENIX Internet Measurement Conference (IMC)* (2005).
- [2] Arbor Networks. <http://www.arbor.com>.
- [3] B. CLAISE, E. Cisco Systems NetFlow Services Export Version 9. RFC 3954, 1990.
- [4] BARFORD, P., KLINE, J., PLONKA, D., AND RON, A. A Signal Analysis of Network Traffic Anomalies. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop (IMW)* (2002).

- [5] BRUTLAG, J. D. Aberrant Behavior Detection in Time Series for Network Monitoring. In *Proceedings of USENIX Large Installation Systems Administration Conference (LISA)* (2000).
- [6] BURCH, H., AND CHESWICK, B. Tracing Anonymous Packets to Their Approximate Source. In *Proceedings of USENIX Large Installation Systems Administration Conference (LISA)* (2000).
- [7] CASE, J., FEDOR, M., SCHOFFSTALL, M., AND DAVIN, J. A Simple Network Management Protocol (SNMP). RFC 1157, 1990.
- [8] Cisco Guard. <http://www.cisco.com/en/US/products/ps5888/>.
- [9] COOKE, E., JAHANIAN, F., AND MCPHERSON, D. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *Proceedings of USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)* (2005).
- [10] DUFFIELD, N., LUND, C., AND THORUP, M. Learn more, sample less: Control of volume and variance in network measurement. *IEEE Transactions in Information Theory* 51, 5 (2005), 1756–1775.
- [11] ESTAN, C., SAVAGE, S., AND VARGHESE, G. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *Proceedings of ACM SIGCOMM* (2003).
- [12] GIL, T., AND POLETTI, M. MULTOPS: A Data-structure for Bandwidth Attack Detection. In *Proceedings of USENIX Security Symposium* (2001).
- [13] JAIN, A., HELLERSTEIN, J. M., RATNASAMY, S., AND WETHERALL, D. A Wakeup Call for Internet Monitoring Systems: The Case for Distributed Triggers. In *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)* (2004).
- [14] JUNG, J., KRISHNAMURTHY, B., AND RABINOVICH, M. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In *Proceedings of the International World Wide Web Conference (WWW)* (2002).
- [15] KANDULA, S., KATABI, D., JACOB, M., AND BERGER, A. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *Proceedings of USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)* (2005).
- [16] KEROMYTIS, A., MISRA, V., AND RUBENSTEIN, D. SOS: Secure Overlay Services. In *Proceedings of ACM SIGCOMM* (2002).
- [17] KOMPELLA, R. R., SINGH, S., AND VARGHESE, G. On Scalable Attack Detection in the Network. In *Proceedings of ACM/USENIX Internet Measurement Conference (IMC)* (2004).
- [18] KRISHNAMURTHY, B., MADHYASTHA, H. V., AND SPATSCHECK, O. ATMEN: a triggered network measurement infrastructure. In *Proceedings of International World Wide Web Conference (WWW)* (2005).
- [19] LAKHINA, A., CROVELLA, M., AND DIOT, C. Diagnosing network-wide traffic anomalies. In *Proceedings of ACM SIGCOMM* (2004).
- [20] LAKHINA, A., CROVELLA, M., AND DIOT, C. Mining anomalies using traffic feature distributions. In *Proceedings of ACM SIGCOMM* (2005).
- [21] MAHAJAN, R., BELLOVIN, S., FLOYD, S., IOANNIDIS, J., PAXSON, V., AND SCOTT, P. Controlling high bandwidth aggregates in the network. *ACM SIGCOMM CCR* 32 (2002).
- [22] Mazu Networks. <http://www.mazu.com>.
- [23] MIRKOVIC, J., AND REIHER, P. A Taxonomy of DDoS Attacks and DDoS Defense Mechanisms. *ACM SIGCOMM CCR* 34 (2004).
- [24] MOORE, D., VOELKER, G. M., AND SAVAGE, S. Inferring Internet Denial-of-Service activity. In *Proceedings of USENIX Security Symposium* (2001).
- [25] MOREIN, W. G., STAVROU, A., COOK, D. L., KEROMYTIS, A., MISRA, V., AND RUBENSTEIN, D. Using Graphic Turing Tests to Counter Automated DDoS Attacks Against Web Servers. In *Proceedings of Networking and Distributed Systems Security Symposium (NDSS)* (2005).
- [26] ROUGHAN, M., GREENBERG, A., KALMANEK, C., RUMSEWICZ, M., YATES, J., AND ZHANG, Y. Experience in Measuring Internet Backbone Traffic Variability: Models, Metrics, Measurements and Meaning. In *Proceedings of International Teletraffic Congress (ITC)* (2003).
- [27] SANDERS, T. Cops smash 100,000 node botnet. <http://www.vnunet.com/2143475>, Oct 2005.
- [28] SAVAGE, S., WETHERALL, D., KARLIN, A., AND ANDERSON, T. Practical Network Support for IP Traceback. In *Proceedings of ACM SIGCOMM* (2000).
- [29] SNOEREN, A. C., PARTRIDGE, C., SANCHEZ, L. A., JONES, C. E., TCHAKOUNTIO, F., KENT, S. T., AND STRAYER, W. T. Hash-Based IP Traceback. In *Proceedings of ACM SIGCOMM* (2001).
- [30] THE HONEYNET PROJECT. Know your enemy: Tracking botnets. <http://www.honeynet.org/papers/bots>.
- [31] VASUDEVAN, R., MAO, Z. M., SPATSCHECK, O., AND VAN DER MERWE, J. Reval: A Tool for Real-time Evaluation of DDoS Mitigation Strategies. In *Proceedings of USENIX Annual Technical Conference* (2006).
- [32] XU, K., ZHANG, Z.-L., AND BHATTACHARYA, S. Profiling internet backbone traffic: Behavior models and applications. In *Proceedings of ACM SIGCOMM* (2005).
- [33] YANG, X., WETHERALL, D., AND ANDERSON, T. A DoS-limiting Network Architecture. In *Proceedings of ACM SIGCOMM* (2005).
- [34] ZHANG, Y., GE, Z., GREENBERG, A., AND ROUGHAN, M. Network anomography. In *Proceedings of ACM/USENIX Internet Measurement Conference (IMC)* (2005).
- [35] ZHANG, Y., SINGH, S., SEN, S., DUFFIELD, N., AND LUND, C. Online detection of hierarchical heavy hitters: algorithms, evaluation, and applications. In *Proceedings of ACM/USENIX Internet Measurement Conference (IMC)* (2004).

Notes

¹Despite this tunability, our experiments with a publicly available multi-dimensional analysis tool [11], suggest that this approach is too compute intensive to scale to the data volumes to be analyzed.

²Due to cost and operational constraints commercial vendor detection systems are typically constrained to operate in such a centralized model using feeds near the core of the network.

³Prior work [1] indicates that using 40-50 frequency coefficients can obtain a good predictive model for weekly traffic volume counts.

⁴Our implementation sets a threshold of 2×10^6 bytes every 300 seconds on smart-sampled flow data, which roughly translates into a raw data rate of $\frac{2 \times 10^6 * 500 * 8}{300} \approx 26$ Mbps.

⁵Our implementation counts the number of distinct flows and sets a threshold of 5 flows every 300 seconds, which translates into an absolute data rate of $\frac{5 * 20MB * 8}{300} \approx 2.6$ Mbps.

⁶Due to collection issues we missed data for the second Monday.

⁷The rates for alarms of duration longer than 300 seconds will be lower than the high intensity thresholds of 26 Mbps for the bandwidth attacks, and 2.6 Mbps for SYN/RST/ICMP attacks, due to the rate depreciation we discussed earlier.

Bump in the Ether: A Framework for Securing Sensitive User Input

Jonathan M. McCune Adrian Perrig Michael K. Reiter
Carnegie Mellon University
{jonmccune, perrig, reiter}@cmu.edu

Abstract

We present Bump in the Ether (BitE), an approach for preventing user-space malware from accessing sensitive user input and providing the user with additional confidence that her input is being delivered to the expected application. Rather than preventing malware from running or detecting already-running malware, we facilitate user input that bypasses common avenues of attack. User input traverses a *trusted tunnel* from the input device to the application. This trusted tunnel is implemented using a trusted mobile device working in tandem with a host platform capable of attesting to its current software state. Based on a received attestation, the mobile device verifies the integrity of the host platform and application, provides a trusted display through which the user selects the application to which her inputs should be directed, and encrypts those inputs so that only the expected application can decrypt them. We describe the design and implementation of BitE, with emphasis on both usability and security issues.

1 Introduction

Using security-sensitive applications on current computer systems exposes the user to numerous risks. User-level malware such as keyloggers, spyware, or Trojans often monitor and log every keystroke. Through keystrokes, an adversary may learn sensitive information such as passwords, bank account numbers, or credit card numbers. Unfortunately, current computing environments make such keystroke logging trivial; for example, X-windows allows any application to register a callback function for keyboard events destined for any application. Giampaolo created a simple 100-line C program that is able to capture keyboard input events that the user intended for some other application under X11. Similar vulnerabilities exist in Microsoft Windows, e.g., `RegisterHotKey` and `SendInput` can be used in combination from an application that does not currently have input focus to capture user input to the application with input focus. We conclude that it is desirable to reduce the involvement of the window manager in sensitive I/O activities as much as possible.

Besides the ease of eavesdropping on keystrokes, another serious risk to the user is the integrity of screen content. Malicious applications can easily overwrite any screen area. This introduces the threat that a user cannot trust any content displayed on the screen since it may

originate from a malicious application. Additionally, malicious software that can capture the screen content of running applications may be able to extract valuable secrets (e.g., the user is filling out an electronic tax form with banking software).

In such an environment, it is challenging to design a system that provides the user with guarantees that the correct operating system and the correct application are currently running, and that only the correct application will receive the user's keystrokes. In particular, we would like a computing environment with the following properties:

- The user obtains user-verifiable evidence that the correct OS and the correct application loaded.
- The user obtains user-verifiable evidence that only the correct application is receiving keystroke events.

We designed and prototyped Bump in the Ether (BitE), a system that proxies user input via a trusted mobile device, circumventing the window manager and traditional input paths via a user-verifiable trusted tunnel. We name the system Bump in the Ether because part of the inspiration for this work came from devices referred to as “bumps in the cord”, which were devices used to “scramble” conversations on plain old telephone service (POTS) in an effort to foil eavesdroppers.

Tunnels in BitE are end-to-end encrypted, authenticated tunnels between a trusted mobile device and a particular application on the user's host platform. Figure 1 shows a comparison of the legacy input path versus input through a trusted tunnel. To reduce the user's need to trust the window manager, we use the display on a trusted mobile device as a trusted output mechanism. We design an OS module that directly passes sensitive keystrokes from the user's mobile device to the correct application, bypassing the X-windowing system.

We assume the user's computing platform is capable of attesting to its current software state. For the remainder of this paper, we assume the user's computing platform is equipped with a Trusted Platform Module (TPM) as specified by the Trusted Computing Group (TCG), and that the BIOS and OS are TPM-enabled and perform integrity measurements of code loaded for execution [26, 37]. The user's mobile device is used to verify these measurements.

Trusted tunnels use per-application cryptographic keys which are established during an application registration phase. The process of establishing a trusted input

session over the trusted tunnel is contingent on the mobile device's successfully verifying an attestation from the integrity measurement architecture (IMA [26]) and TPM on the user's host platform, as well as the user selecting the correct application from a list presented by her mobile device.

2 Background and Related Work

We provide background and review related work on mobile devices, secure window managers, and trusted computing primitives.

2.1 Mobile Devices

Balfanz and Felten explored the use of hand-held computing devices (e.g., PDAs) as smart-cards, and found some advantages because the user can interact directly with the hand-held for sensitive operations [1]. The authors generalized their work into a design paradigm they call *splitting trust*, where a smaller, trusted device performs security-sensitive operations and a large, powerful device performs other operations. BitE can be considered a system designed in accordance with the principles of splitting trust.

The Pebbles project attempts to let handhelds and PCs work together when both are available, as opposed to the conventional view that handhelds are used when PCs are unavailable [20]. BitE uses a trusted mobile device to help improve input security on a PC.

Ross et al. develop a framework for access to Internet services, where both the sensitivity of the information provided by the service and the capabilities of the client device are incorporated [25]. This framework depends on either a trusted proxy infrastructure or service providers running a trusted proxy. While promising, this scheme is not widely deployed today.

Sharp et al. develop a system for splitting input and output across an untrusted terminal and a trusted mobile device [31]. Applications run on a trusted server or on the mobile device itself, using VNC [23] to export video to the trusted and untrusted displays in accordance with a security policy. The user has the ability to decide on the security policy used for the untrusted keyboard, mouse, and display. An initial user study yielded encouraging results, but this technique is best described as a tool for power users. In contrast, BitE is designed for interaction with applications running on a local workstation, and for users who may have very little understanding of computer security.

Sharp et al. propose an architecture for fighting crime-ware (e.g., keyloggers and screengrabbers) via split-trust web applications [30]. Web-applications are written to support an untrusted browser and a trusted mobile device with limited browsing capabilities. All security-critical decisions are confirmed on the mobile device. This architecture raises the bar for web-based attackers, but it also raises usability issues which are the subject of future

work. BitE is designed to improve the security of interaction between a user and applications on her local workstation. While that application can be a web browser, BitE does not specifically address web security issues.

2.2 Secure Window Managers

A goal of BitE is to ensure that only the correct application is receiving input events, and to provide user-verifiable evidence that this is so. While much prior work has addressed this issue, none of it is readily available for non-expert users on commodity systems today. We now review related work chronologically.

Several government and military computer windowing systems have been developed with attention to security and the need to carefully isolate different grades of information (e.g., classified, secret, top secret). Early efforts to secure commercial window managers resulted in the development of *Compartmented Mode Workstations* [5, 6, 11, 22, 24, 38], where tasks with different security requirements are strictly isolated from each other. These works consider an operating environment where an employee has various tasks she needs to perform, and some of her tasks have security requirements that necessitate isolation from other tasks. For example, Picciotto et al. consider trusted cut-and-paste in the X window system [21]. Cut-and-paste is strictly confined to allow information flow from low-sensitivity to high-sensitivity applications, so that high-sensitivity information can never make its way into a low-sensitivity application. Epstein et al. performed significant work towards trusted X for military systems in the early 1990s [8, 9, 10]. While these systems are effective for employees trained in security-sensitive tasks, they are unsuitable for use by consumers.

Shapiro et al. propose the EROS Trusted Window System [29], which demonstrates that breaking an application into smaller components can greatly increase security while maintaining very powerful windowing functionality. Unfortunately, EROS is incompatible with a significant amount of legacy software, which hampers widespread adoption. In contrast, BitE works in concert with existing window managers.

Microsoft's Next-Generation Secure Computing Base (NGSCB) proposes encrypting keyboard and mouse input, and video output [18]. In NGSCB, special USB keyboards encrypt keystrokes which pass through the regular operating system into the *Nexus*, where they are decrypted. Once in the *Nexus*, they can be sent to a trusted application running in *Nexus-mode*, or they can be sent to the legacy OS. Applications running in *Nexus-mode* have the ability to take control of the system's primary display, which was designed to be useful for establishing a trusted tunnel.

Common to the majority of these schemes is a mechanism by which some portion of the computer's screen is trusted. That is, an area of the screen is controlled by

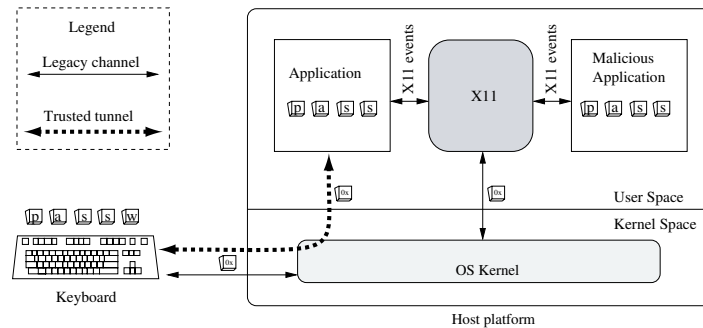


Figure 1: Traditional flow of keystrokes vs. trusted tunnels. On a traditional computer system, keystrokes are first sent to the OS kernel, which passes them to X-windows, which then sends keyboard events to all applications that register to receive them. Unfortunately, malicious applications can register a callback function for keyboard events for other applications. Our trusted tunnels protect keystrokes and only send them to the desired application.

some component of the trusted computing base (TCB) and is inaccessible to all user applications. However, if an application can use a “full-screen” mode, it may be able to spoof any trusted output. Precisely defining trusted full-screen semantics that a non-expert user can operate securely is, to the best of our knowledge, an unsolved problem. Considering the value that the user receives from being able to maximize applications, and the role of multi-media applications on today’s commodity PCs, we believe the ability to run applications in full-screen mode on the system’s primary display is an indispensable feature. Still, there is no effective way to establish a trusted tunnel if there is no trusted display. Due to the complexity of X and the likely confusion of untrained users, it is difficult to implement a trusted screen area in an assurable way. BitE uses the trusted mobile device’s screen—a physically separate display—as a trusted output device.

We emphasize that, despite the large body of work on trusted windowing systems, the majority of users do not employ any kind of trusted windowing system. Thus, we proceed under the assumption that users do not want to change their windowing system. In the remainder of the paper, we show that BitE can increase user input security under these conditions.

2.3 Trusted Computing Primitives

BitE leverages two features of the user’s TPM-equipped computing platform: attestation and sealed storage. To enable these features, the platform’s OS must be equipped with an integrity measurement architecture (IMA). The values resulting from integrity measurement are used in attestations and in access control for sealed storage. The remainder of this section provides more detail on these mechanisms.

TPMs have 16 platform configuration registers (PCRs) that an IMA can extend with *measurements* (typically cryptographic hashes computed over a complete executable) of software loaded for execution. The IMA extends the appropriate PCR registers with the measure-

ment of each software executable just before it is loaded.

TPMs can generate a Storage Root Key (SRK) that will never leave the chip. The SRK enables *sealed storage*, whereby data leaving the TPM chip is encrypted under the SRK for storage on another medium. Data can be sealed with respect to the values of certain PCR registers, so that the unsealing process will fail unless the same values are present that were present during the sealing process. Several other keys are maintained by the TPM and kept in sealed storage when not in use. One of these is the Attestation Identity Key (AIK), which is an RSA signing keypair used to sign attestations. To the remote party trying to verify the attestation, the public AIK represents the identity of the attesting platform.

An *attestation* produced by the IMA and TPM consists of two parts: (1) a list of the measurements of all software loaded for execution, maintained by the IMA functionality in the OS; and (2) an AIK-signed list of the values in the PCR registers, called a *PCR quote*. A remote party with an authentic copy of the public AIK can compute the expected values for the PCR registers based on the measurement list, and check to see whether the signed values match the computed values. The end result is a *chain* of measurements of all software loaded since the last reboot. The security requirement is that all software is measured before being loaded for execution.

Sailer et al. developed an open-source IMA for Linux [26]. They show that it is difficult to manage the integrity measurements of a complete interactive computer system, since there will be hundreds or even thousands of measurements in the course of a normal system’s uptime, and the order in which applications are executed is reflected in the resulting PCR values. This ordering is not a problem for attestation, since the measurement list can be validated against the PCR values. However, it is an issue for sealed storage, since data is sealed with respect to PCR values, and not particular items on the measurement list. Thus, to access data in sealed storage, not only must the same software be loaded, but it must have been loaded in the *same order*.

During the boot process, however, a well-behaved system always loads in the same order. Hence, integrity measurement of the system from boot through the loading of the kernel, its modules, and system services loaded in a deterministic order will be consistent across multiple boot cycles on a well-behaved host platform.

One problem with IMA as described is that integrity measurements are made at *load-time*. Thus, run-time vulnerabilities may go undetected if malicious parties can exploit the difference between time-of-check and time-of-use (so-called TOCTOU attacks). IMA can be used in combination with systems which provide run-time attestation [32] or verifiable code execution [28] to achieve even stronger platform security guarantees.

3 Bump in the Ether

We provide a brief overview of BitE and present the assumptions under which it operates. We then discuss the necessary setup which must take place before BitE can be used. We describe the use of BitE to secure user input in Section 4.

3.1 System Overview

The primary goal of BitE is to enable end-to-end encrypted, authenticated input between the user's trusted mobile device and an application running on her host platform. BitE is built around a trusted mobile device that can proxy user input, show data on its own display, efficiently perform asymmetric and symmetric cryptographic operations, and store cryptographic keys. This trusted device runs a piece of software called the BitE Mobile Client. The BitE Mobile Client verifies attestations from the host platform, manages cryptographic keys used in establishing trusted tunnels for user input, and provides a trusted output mechanism which can inform the user of security-relevant events.

We summarize the two setup steps and common usage for BitE.

1. Building an association between the trusted mobile device and the host platform (Section 3.3.1). This is performed once for each trusted mobile device and host platform pair.
2. Registering applications on the host platform for use with BitE (Section 3.3.2). This is performed once for each registered application on a particular trusted mobile device and host platform pair.
3. Sending user input to a registered application (Section 4). This is performed every time secure input to a registered application is required. We consider two kinds of registered applications, those which are BitE-aware, and those which are not.

3.2 Threat Model and Assumptions

Users use their computers to process sensitive information, for example, banking applications, corporate VPNs, and management of financial information. Attackers are

interested in stealing such information, often with the intent to commit identity theft. One technique which attackers use is user-space malware, including Trojans and spyware such as keyloggers and screengrabbers.

BitE protects user input against user-space malware. We assume attackers are capable of passive monitoring and active injection attacks on the network link between the user's trusted mobile device and her host platform. We assume the user's mobile device is not compromised, although we discuss the possibility of using mutual attestation between the mobile device and host platform to detect a compromised mobile device in Section 5.3.

The mobile device must simultaneously connect to the user's input device (e.g., keyboard) and her host platform. We assume that a secure (direct physical connection, or authenticated and encrypted wireless connection) association between the user's input device and her mobile device can be established. However, there are environments where the user is wary of trusting an unknown keyboard for fear of hardware keyloggers. In such environments, the user can enter sensitive information directly into her mobile device, avoiding the use of the suspicious keyboard. Physical attacks such as "shoulder surfing" and keyboard emanation attacks [41] are beyond the scope of BitE. Thus, we do not discuss them further. For the remainder of the paper, we consider the user's input device (e.g., wireless keyboard) as an extension of her mobile device. Subsequent discussions will focus on interaction between the user, her mobile device, and her host platform.

We use attestation to verify the integrity of code loaded for execution on the user's host platform, so we need *not* assume that the host platform's software integrity is intact. However, as discussed in Section 2.3, the integrity measurement architecture used for attestation has TOCTOU limitations. Specifically, attestation allows us to detect modified program binaries before they are loaded. If a loaded application is compromised while running, however, IMA will not detect it.

We must trust the OS kernel on the host platform with which the user wishes to establish a trusted tunnel for input. One reason we must trust the OS kernel is because of its ability to arbitrarily read and modify the memory space of any process executing on the system—we cannot trust an application without also trusting the kernel on which it runs.

3.3 BitE Setup

We now describe application setup with BitE, which consists of two steps: (1) an association between the trusted mobile device and the host platform must be created, and (2) applications for which BitE will be used to secure their input must be registered with the BitE system.

3.3.1 Device Association

An association between the trusted mobile device and the host platform consists of two parts: (1) the BitE Mobile Client's ability to verify attestations from the host platform, and (2) keys for mutually authenticated and encrypted communication between the BitE Mobile Client and the BitE Kernel Module. The purpose of the attestation is to detect malware on the host platform; the purpose of the mutually authenticated and encrypted communication is to thwart active injection and passive monitoring attacks on the wireless connection between the BitE Mobile Client and the BitE Kernel Module.

To enable the BitE Mobile Client to verify attestations, it must be equipped with a set of expected measurements of acceptable software configurations for the host platform. However, the set of all possible software configurations that may be running on the host platform is unmanageable, as that set may include any piece of software that the user installs (willingly or unwillingly) on her host platform.

Our solution is to verify the measurements of those software components which are expected to change infrequently on a healthy system: the boot stack, the kernel, its modules (including the BitE Kernel Module), and well-ordered system services (those software components whose measurements are expected to occur in the same order between boot cycles). The BitE Mobile Client must be equipped with the expected values for the measurements of this infrequently-changing software so that it can verify attested measurements. In our current prototype, this is a manual process by which we add the expected filenames and measurements to a configuration file on the BitE Mobile Client. Another solution would be to add a configuration program which runs last during the boot process on the host platform, so that it can save a copy of all measurements of well-ordered services. Note that this still has the drawback that we are simply assuming the system to be secure when the configuration program runs; this is best done on a new system before it is connected to the Internet or other potential source of malware.

To authenticate the origin of an attestation, and to verify that the measurement list received matches the PCR quote from the host platform's TPM, the BitE Mobile Client must be equipped with the public *Attestation Identity Key* (*AIK*) from the host platform's TPM. The *AIK* is required by the BitE Mobile Client to verify the digital signature on attestations from the TPM in the host platform. Our current design only performs one-way attestation (host platform to mobile device); however, we discuss the possibility of mutual attestation in Section 5.3.

The BitE Mobile Client and the BitE Kernel Module must be able to establish mutually authenticated, encrypted communication to resist active injection and passive monitoring attacks on the wireless link between the user's mobile device and her host platform. Standard

protocols exist for this purpose (e.g., SSL [12]) provided that authentic cryptographic keys are in place. We use the notation $\{K_{KM}, K_{KM}^{-1}\}$, $Cert_{KM}$ and $\{K_{MC}, K_{MC}^{-1}\}$, $Cert_{MC}$ for the asymmetric (e.g., RSA) keypairs and certificates (e.g., self-signed X509) for the BitE Kernel Module and the BitE Mobile Client, respectively.

We assume for simplicity that the attacker is not present during the exchange of *AIK*, $Cert_{KM}$, and $Cert_{MC}$. Thus, our current prototype exchanges these keys in the clear. We note that securing this key exchange is possible, though challenging. A potential solution is to use location-limited side channels to exchange pre-authentication data. Seeing-is-Believing and related techniques could be applied here [2, 16, 34].

3.3.2 Application Registration

Applications must be registered with the BitE Kernel Module and the BitE Mobile Client so that their integrity can be verified during subsequent attempts by the user to send input to them. The integrity measurement of each application serves as access control for application-specific cryptographic keys which are used to establish the trusted tunnel for user input.

To obtain the expected measurement value of an application, the user first indicates her desire to register a new application to the BitE Mobile Client. She then performs an initial execution of the application to be registered on her host platform. The IMA automatically measures this application and its library dependencies and stores them in the IMA measurement list (see [26] for details). We assume the system state can be trusted during application registration (i.e., there is no malicious code executing). Note that other dependencies may exist that we wish to measure. For example, configuration files can have a significant impact on application security. Automatic identification of configuration files associated with a particular application is complex, and beyond the scope of this paper. Alternatively, a trusted authority could provide the measurement values for a trustworthy version of the application.

The BitE Kernel Module generates a symmetric key K_{App_i} (for application *i*) to be used in subsequent connections for the derivation of encryption and MAC session keys for establishing the trusted tunnel. These per-application cryptographic keys are kept in TPM-protected sealed storage [37]. They are sealed under the PCR values which represent the boot process up through the loading of well-ordered system services (as described in Section 3.3.1). Handling which PCR registers receive which measurements is an issue which requires some care. We introduce the issue, and then present two possible solutions.

We dedicate a subset of the available PCRs for measurements of the well-ordered system services. However, once the appropriate measurements are in these PCRs, the TPM will allow the secrets in sealed storage to be

unsealed. A tempting solution is to invalidate these PCRs after the BitE Kernel Module reads the sealed secrets by “measuring” some random data; however, this prevents the BitE Kernel Module from adding new secrets to sealed storage during the current boot cycle.

One option is to use dedicated PCR registers for the well-ordered system services, but not invalidate these registers after the BitE Kernel Module reads the sealed secrets. We must then trust the OS to restrict access to the TPM and hence the interface with sealed storage. This way, the BitE Kernel Module can add new secrets to sealed storage at any time. Our current prototype uses this option, which is more convenient, but less secure, than the next option.

An alternative is to allow application registration only immediately after a reboot. Thus, the application to be registered is the only software run during that boot cycle that is not part of the well-ordered system services, resulting in a more secure system state for registration, so that it is safer to leave the contents of the PCRs dedicated to well-ordered system services intact. Note that the reboot into “registration mode” may need to be a special, reduced-service reboot if there is non-determinism in the order in which some services are executed. For example, a host platform which starts a web server early in the boot process may execute a CGI script before the rest of the system has finished booting and invalidate some PCR values.

The IMA measurement for application i , the newly generated symmetric key K_{App_i} , and the user-friendly name of the registered application (e.g., *OpenOffice Calc*), are sent over a mutually authenticated, encrypted channel (established using $Cert_{KM}$ and $Cert_{MC}$ in, e.g., SSL with ephemeral Diffie-Hellman key agreement [12]) to the BitE Mobile Client, where they are stored for future use.

4 Operation

We now describe the actual process by which a user using BitE securely enters input into a registered application. At this point, the association between the trusted mobile device and the host platform is established, and the user’s application(s) have been registered with the BitE system. The goal is to establish an end-to-end trusted tunnel for input between the user’s trusted mobile device and an application running on her host platform. We define a trusted tunnel to be a mutually authenticated, encrypted network connection. We describe BitE using Linux and X11 terminology [39, 40]. However, our techniques can be applied to other operating environments, e.g., Microsoft Windows or Apple OS X, contingent on the existence of some means for attesting to the current software state.

We first consider applications designed with knowledge of BitE (Section 4.1), and later describe a wrapper which can be used with legacy applications (Section 4.2).

We also consider conflict resolution which becomes necessary when multiple applications require a trusted tunnel concurrently (Section 4.3). Figure 2 shows a procedural overview of secure input using BitE.

4.1 BitE-Aware Applications

Establishment of a trusted tunnel is initiated by a BitE-aware application when it requires sensitive input (e.g., passwords or credit card numbers) from the user. The application sends a message to the BitE Kernel Module to register an input-event callback function implemented by the BitE-aware portion of the application. If the BitE Kernel Module has no other outstanding requests, it begins the process of establishing the trusted tunnel. We discuss extensions to allow the user to manually initiate a trusted tunnel in Section 6.2.

There are three steps in the establishment of a secure input session:

1. Verification by the BitE Mobile Client of an attestation produced by the host platform, including verification that the desired application was loaded (Section 4.1.1).
2. Interaction between the user and her trusted mobile device to confirm that it is indeed her desired application which is requesting secure input (Section 4.1.2).
3. Establishment of the session keys which will be used to encrypt and authenticate the actual keystrokes entered by the user (Section 4.1.3).

4.1.1 Attestation of Host Platform State

The BitE Mobile Client must verify an attestation of the software which has been loaded on the user’s host platform. This serves two purposes: (1) to ensure the the the well-ordered system services on the host platform have not been modified since the mobile device / host platform association, and (2) to ensure that the trusted tunnel for input is established with exactly the same application that was initially registered. The BitE Mobile Client can verify the signature on the attestation with its authentic copy of the public AIK, and it can verify that the measurement list is consistent with the signed PCR quote. It can then compare the measurement values with those present during application registration (Section 3.3.2). If the values match, we consider the host platform to have successfully attested its software state to the BitE Mobile Client. The trusted mobile device now has assurance that the loaded versions of the well-ordered system services and the user’s desired application match those recorded during registration. If this verification fails, then the process of establishing a trusted tunnel for input is aborted, and the user is notified via the display on her trusted mobile device.

Note that the software state of a host platform is comprised of *all* software loaded for execution; we discussed the difficulty of managing this state space in Section 2.3.

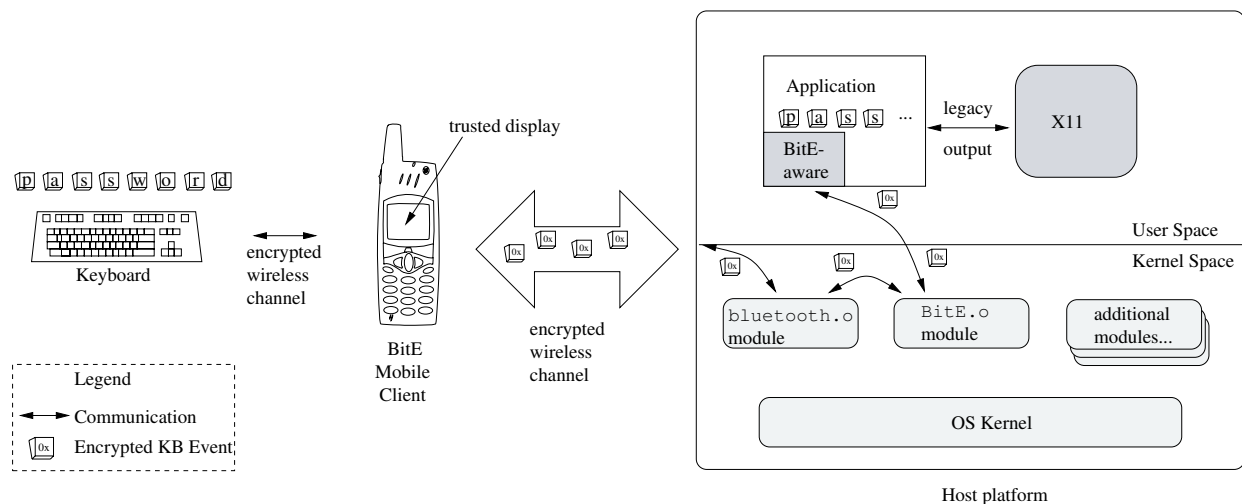


Figure 2: BitE system architecture. The user presses keys (e.g., types a password) on the keyboard. The keypress events are sent over an encrypted channel to the BitE Mobile Client. The BitE Mobile Client re-encrypts the keyboard events with a cryptographic key that is specific to some application. On the host platform, the encrypted keyboard events are passed to the BitE Kernel Module, and then to the application, where they are decrypted.

We verify measurements of the well-ordered system services and the user’s desired application, but other software may be executing. It is such unknown user-level software against which the trusted tunnel offers protection.

4.1.2 User / Mobile Device Interaction

The trusted mobile device’s display serves as a trusted output channel to the user. This enables us to minimize the amount of trust we place in the window manager on the host platform. Upon verifying an attestation from the host platform, the BitE Mobile Client has assurance that the correct application was loaded. Before session keys can be established to form the trusted tunnel, it is necessary to involve the user via her trusted mobile device to ensure that the application with which the user intends to interact and the application asking for her input are the same. This property can be challenging to achieve without annoying the user. A viable solution is one that is easy to use, but not so easy that the user “just hits OK” every time.

Our solution is to display a list of registered applications on the BitE Mobile Client. The user must scroll down (using the arrow keys on her keyboard, or navigational buttons on the trusted mobile device itself) and then select (e.g., press enter) the correct application. Note that since all input from the user’s keyboard passes through the trusted mobile device the user does not actually need to press buttons on the mobile device. The mobile device will interpret the user input from her key-

board appropriately. We believe user confusion will be minimal, but a user-study is needed to validate this solution. Refer to Figure 3 for more information on the interaction between the user and her trusted mobile device.

We are concerned about users developing habits that might increase their susceptibility to spoofing attacks. Thus, we randomize the order of the list so that the user cannot develop a habit of pressing, e.g., “down-down-enter,” when starting a particular application that requires a trusted tunnel. Instead, the user must actually read the list displayed on her mobile device and think about selecting the appropriate application. We believe selection from a randomized list achieves a good balance between security and usability, provided that the length of the list is constrained (for example, that it always fits on the mobile device’s screen).

Once the list is displayed, the BitE Mobile Client signals the user, e.g., by beeping. This serves two purposes: (1) to let the user know that a secure input process is beginning; and (2) to let the user know that she must make a selection from choices on the mobile device’s screen. Item (2) is necessary because a user may become confused if her application seems unresponsive when in reality the BitE Mobile Client on her mobile device is prompting her for a particular action.

Note that a look-alike (e.g., Trojan, spoofing attack) application will be unable to get the mobile device to display an appropriate name, because the look-alike application was never registered with the BitE system. Only

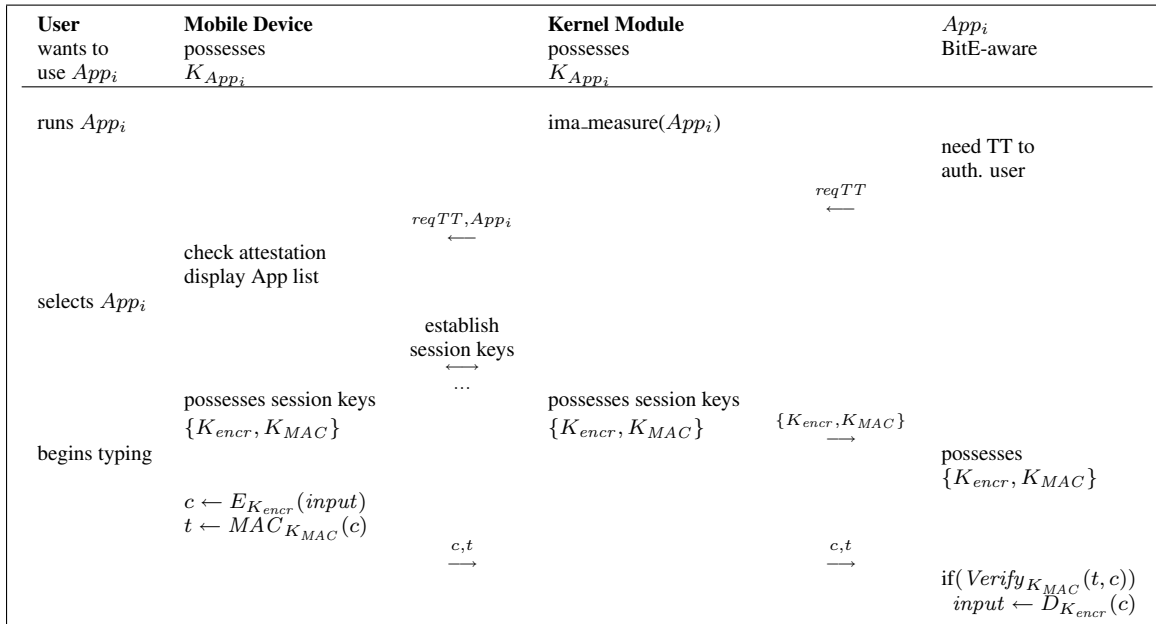


Figure 3: Simplified application execution and trusted tunnel establishment with registered BitE-aware application so that the user can enter sensitive information to that application. In this figure we assume the user's input device(s) is an extension of the trusted mobile device, so we do not show input device(s). We also assume host platform / BitE Mobile Client association (Section 3.3.1) and application registration (Section 3.3.2) have already been successfully completed. *input* consists of padding, a sequence number, and the actual input event. TT = Trusted Tunnel.

applications that were initially registered are options for trusted tunnel endpoints.

If the user is satisfied, she selects the option given by her mobile device corresponding to the name of the application with which she wants to establish a trusted tunnel. If she suspects anything is wrong, she selects the *Abort* option on her mobile device. It is an error if the user selects any application other than the one which is currently requesting a trusted tunnel. That is, the BitE Mobile Client will report an error to the user (the application she selected from the list is not the same application that requested a trusted tunnel). It is a policy decision to decide how to handle this type of error. One approach is to fail secure, and prevent the user from entering sensitive input into her application until a successful retry.

Variations on this user interface that might also be effective in practice are discussed in Section 6.2.

4.1.3 Session Keys

At this point, the BitE Mobile Client has verified an attestation from the host platform, proving that the desired application and the correct well-ordered system services have been loaded. In addition, the user has selected the same application on her mobile device that requested secure input. To complete the encrypted, authenticated tunnel for input, session keys must be established.

Keys are established for encrypting and authenticating user input-related communication to and from the BitE Mobile Client. Even though user input is a one-way con-

cept, bidirectional communication is necessary to properly support complex key sequences such as auto-repeat and Shift-, Control-, and Alt- combinations. The session keys are derived from the per-application keys established during application registration using standard protocols [17].

The BitE Mobile Client uses the session keys to encrypt and MAC the actual keyboard events such that they can be authenticated and decrypted by the BitE-aware application in an end-to-end fashion. Our current prototype does not consider keystroke timing attacks [33, 35]; incorporation of countermeasures for such attacks is the subject of future work.

Figure 3 presents step-by-step details on the process of input via the BitE trusted tunnel. For simplicity, the figure shows user input as a one-way data flow. K_{encr} and K_{MAC} are the encryption and MAC keys for input data flowing from the BitE Mobile Client to the application.

Once the trusted tunnel is established, the user can input her sensitive data. When the application is finished receiving sensitive input, it notifies the BitE Kernel Module that it is finished receiving input via the trusted tunnel. At this point, the BitE Kernel Module tears down the encrypted channel from the BitE Mobile Client to the application, and reverts to listening for requests for trusted tunnels from other registered applications. The BitE Mobile Client notifies the user that the secure input session has finished.

4.2 BitE-Unaware (Legacy) Applications

We now describe BitE operation with an application that is *unaware* of the BitE system. That is, this section describes how BitE is backwards-compatible with existing applications. Legacy applications were written without knowledge of BitE, so there is no way for a legacy application to request a trusted tunnel. Hence, all input to a legacy application must go through a trusted tunnel. The basic idea is that we run legacy applications inside a *wrapper* application (the BitE-wrapper) that provides input events to that application (e.g., `stdin` or X keyboard events).

The legacy application is automatically measured by the IMA, and it is registered with BitE in the same way BitE-aware applications are registered. If the application changes after its initial registration, the BitE Kernel Module will not release the application-specific keys necessary to establish session keys for encrypting and authenticating keyboard events. Note that the application-specific keys can be unsealed from sealed storage as long as the measurements of the well-ordered system services are as expected. Once the BitE Kernel Module has access to the application's keys, it will only release them to the application if that application's measurement matches the expected value.

The most challenging part of interacting with a legacy application is that it contains no BitE-aware component that can handle the decryption and authentication of keyboard events. Instead, the BitE-wrapper does the decryption and authentication of keyboard events. It is necessary to prevent the legacy application from receiving keyboard events from the window manager (or other user-level processes), while allowing it to receive input from the wrapper application. This is easy to achieve for console applications (e.g., just redirect `stdin`); however, it is challenging for graphical applications. We now consider the necessary BitE-wrapper functionality for X11 applications.

X11 applications (*clients* in the context of X) register to receive certain types of event notifications from the X server. Common event types include keyboard press and release events. Applications register to receive these events using the `XSelectInput` function. The BitE-wrapper application can intercept this call for dynamically linked applications using the `LD_PRELOAD` environment variable. With `LD_PRELOAD` defined to a custom BitE shared library, the run-time linker will call the BitE `XSelectInput` instead of the X11 `XSelectInput`. Thus, the BitE Kernel Module is hooked into the application's input event loop. The BitE Kernel Module can generate its own input events to send to the application simply by calling the callback function that the application registered when it called `XSelectInput`.

4.3 Handling Concurrent Trusted Tunnels to Prevent User Confusion

While there are no technical difficulties involved in maintaining multiple active trusted tunnel connections from the BitE Mobile Client to applications, there are user-interface issues. We know of no way to disambiguate to the user which application is receiving input without requiring user diligence. For example, a naive solution is to display the name of the application for which user input is currently being tunneled on the mobile device's screen. This requires the user to look at the screen of her mobile device and ensure that the name matches that of the application with which she is currently interacting.

To prevent user confusion, we force the user to interact with one application at a time in a trusted way. If we allow users to rapidly switch applications (as today's window managers do), then the binding of user intent with user action is dramatically weakened. The rapid context switching makes it easy for the user to become confused and enter sensitive input into the wrong application. An adversary may be able to exploit this weakness. Some users may find this policy annoying; we discuss an alternative policy in Section 6.2.1.

We consider two example applications which we assume to be BitE-aware and that require a trusted tunnel for user authentication:

1. Banking software which requires the user to authenticate with an account number and a password.
2. A virtual private network (VPN) client which requires the user to authenticate with a username and a password.

Suppose the user needs to interact with both applications at the same time, for example, to compare payroll information from her company with entries in her personal bank account. In today's systems, there is nothing to cause the user to serialize her authentication to these applications. She may start the banking software, then start the VPN client, then authenticate to the banking software, then authenticate to the VPN client. In the BitE system, assuming the banking software and VPN client are BitE-aware, the BitE Kernel Module considers this behavior to be a concurrent request by two applications for establishment of a trusted tunnel.

It is a policy decision how to handle concurrent trusted tunnel requests. One option is to default-deny both applications, and alert the user to the contention. She can then retry with one of the two applications, and use it first. This forces the user to establish a trusted tunnel to the first application and fully input her sensitive data to that application. Once her data is input, the first application will relinquish the trusted tunnel, and it will be torn down by BitE. The user can then begin the process of entering her sensitive data to the second application, which will entail the establishment of another trusted tunnel. These one-at-a-time semantics may induce some additional la-

tency for the user before she can begin using her applications, but we consider this to be an acceptable tradeoff in light of the gains in security.

5 Security Analysis

In this section we analyze the security of BitE. During the design of BitE, we tried to make it difficult for the user to make self-destructive mistakes. For example, the BitE Mobile Client will not allow the user's keystrokes to reach the BitE Kernel Module if verification of an attestation fails. The user must respond to messages displayed on her mobile device before she can proceed. Security mechanisms on the critical input path cannot go unnoticed by the user. These mechanisms must provide value commensurate with the difficulty of using them.

We provide some examples of attacks that BitE is able to protect against. We then consider the failure modes of BitE when the assumptions upon which it is constructed do not hold.

5.1 Defensed Attacks

We consider multiple scenarios where the use of BitE protects the user.

Capturing Keystrokes with X Giampaolo shows how easy it is for an attacker to use a malicious application to capture the keystrokes the user intends to go to the active (and assumed benign) application. If the user is using BitE to enter sensitive data, however, this attack does not work (see Figure 1). The user's keystrokes are encrypted and authenticated with session keys (as discussed in Section 4) which are unavailable to the malicious application. Hence, the encrypted keystrokes reach the user's desired application unobserved.

Software Keyloggers Software keyloggers are a significant threat. Sumitomo Bank in London was the victim of a sophisticated fraud scam involving software keyloggers [27]. With BitE, the user's keystrokes travel inside encrypted, authenticated tunnels. Even if an adversary can capture the ciphertext, he will be unable to extract the keystrokes.

Bluetooth Eavesdropping BitE is most convenient for the user when wireless communication mechanisms can be used between the mobile device and the host. As long as the initial exchange of public keys between the BitE Mobile Client and the BitE Kernel Module proceeds securely, all communication between them can be encrypted and authenticated using standard protocols. If keystroke timing attack countermeasures are incorporated [33, 35], timing side-channels can also be eliminated. Since all communication is strongly authenticated, an adversary will not be able to masquerade as a valid BitE Mobile Client or BitE Kernel Module.

Modification of Registered Applications An attacker may be able to modify (e.g., by exploiting a buffer overflow vulnerability in a different application) the binary of a registered application. Such an attack may modify

the application's executable such that it may log user input to a file, or send it to a malicious party on the Internet. With BitE, an IMA measurement of the executable was recorded during initial application registration (recall Section 3.3.2). The modified application binary will be detected during trusted tunnel setup when the BitE Mobile Client tries to verify the attestation from the host platform. The BitE Mobile Client will alert the user that the application has been modified.

Kernel Modification A measurement of the kernel binary is part of the integrity measurement which is verified when a trusted tunnel is established. Modification of the kernel image on disk will be detected after the next reboot. As a disk-only modification of the kernel image will not affect the running system until a reboot, the attack is detected by the BitE Mobile Client before it can affect the operation of BitE.

5.2 Failure Modes

We now describe what happens if the assumptions upon which the security of BitE is based turn out to be invalid. Specifically, we discuss the extent to which the failure of our assumptions permit the attacker to perform one or more of the following:

- To observe keystrokes in an ongoing session.
- To observe keystrokes in current and future sessions.
- To register malicious applications.

Compromise of Active Application If the attacker is able to compromise an application while the user has a trusted tunnel established, he may be able to observe the user's keystrokes. This break is limited to the compromised application, however, as the attacker has no way to access keys established between the BitE Kernel Module and other registered applications. This break is feasible because the adversary is exploiting a time-of-check, time-of-use (TOCTOU) limitation of the integrity measurement architecture (e.g., a buffer overflow attack). Incorporation of mechanisms for run-time attestation (e.g., Pioneer [28]) can help defend against this attack.

Compromise of Active Kernel on Host Platform If the operating system kernel on the host platform is compromised without rewriting a measured binary (e.g., exploiting TOCTOU limitations with a buffer overflow attack), the attacker may be in a position to capture sensitive user input despite the BitE system. This gives the attacker access to K_{KM}^{-1} and to the unique application keys K_{App_i} . The attacker can also capture keystrokes from ongoing sessions by reading the session keys out of the memory space of the BitE Kernel Module or the applications.

Compromised Mobile Device Since the mobile device is used as a central point of trust in our system, its compromise will allow an attacker to access all keyboard events. The attacker will have possession of K_{MC}^{-1} so he may be able to masquerade as a trusted BitE Mobile

Client using an arbitrary device, such as one with a very powerful radio transmitter. Further, the attacker will capture all registered applications' unique keys, K_{App_i} for application i , and user-friendly name. This will enable the attacker to establish trusted paths with registered applications, and it will allow the attacker to register new applications.

Hardware Keyloggers Malicious parties may use hardware as well as software techniques to record users' keystrokes. BitE is designed to protect against software-based attacks. To protect against hardware-based attacks, the user would need to carry their own keyboard as well as their mobile device, which we consider to be an excessive burden. However, if the amount of sensitive data that the user must enter is small, she can use BitE without an external keyboard and enter her sensitive data directly into her mobile device.

5.3 Mutual Attestation

Currently, there is no attestation technology available for mobile devices. However, the TCG is currently working on trusted platform standards for mobile devices [36], which may be able to minimize the severity of a mobile device compromise. With such technology, it becomes possible to implement mutual attestation in BitE, where the host platform verifies an attestation from the mobile device in addition to the mobile device verifying an attestation from the host platform.

Mutual attestation can increase an attacker's workload, since compromising only the mobile device is no longer a sufficient attack. To fully circumvent BitE, the attacker would have to compromise both the BitE Mobile Client and the user's host platform. For example, the BitE Mobile Client could store its secrets in sealed storage, rendering them inaccessible to malicious software installed on the mobile device by an adversary.

6 Discussion

In this section we discuss additional issues that arise while using the BitE system. These issues include alternative system architectures, e.g., elimination of the trusted mobile device or TPM, and alternative user interface designs for the BitE system.

6.1 Alternative System Architectures

The BitE system as presented in this paper is designed around a TPM-equipped host platform and a trusted mobile device. We briefly consider alternative design approaches, namely, designs that eliminate the mobile device or the TPM. It is particularly tempting to think that one or both of these requirements may be unnecessary since we include the OS kernel in the TCB for BitE. In addition to its role as resource manager, we must trust the OS kernel because of its ability to arbitrarily read the memory space of any process executing on the system. Other researchers have tried to quantify the extent

to which data is exposed by measuring the *lifetime* of data in a system [7, 13]. In BitE, we minimize the data lifetime of user input by minimizing the quantity of code through which cleartext input passes.

6.1.1 Eliminating the Mobile Device

A significant challenge addressed by BitE is for the user to obtain a user-verifiable property of the integrity of the OS and application. It is important to recall two of the roles the mobile device fulfills:

- Verification of integrity measurements from the host platform.
- Trusted visual output to the user.

Integrity measurement on the host platform puts in place the facility for verification by an external entity, but the host platform cannot "self-verify." In BitE, the mobile device fulfills the role of the verifying entity. The mobile device must have trusted visual output to the user so it can appropriately notify the user of the success or failure of this verification.

6.1.2 Input Proxying by the Mobile Device

The OS kernel on the user's host platform is part of the TCB when using BitE. With a trusted OS kernel, there is no technical reason why user input must travel through the mobile device. The BitE Kernel Module already possesses copies of the cryptographic keys shared with the application, and it could encrypt user input from the traditional keyboard driver before it passes through the window manager. However, this design raises an important usability issue. The mobile device must be on the critical input path so that it can ensure that the user cannot proceed unaware of a failed integrity verification. We are concerned that non-expert users will proceed to interact with their application even if a message appears on the mobile device stating that the system is compromised. With the mobile device on the critical input path, it can stop user input from reaching the application while also providing secure feedback to the user.

6.1.3 TPM Alternatives

Finally, we discuss the extent to which the TPM is an essential requirement for BitE. We could leverage software-based attestation mechanisms to verify the authenticity of the OS [28]. However, in open computing environments, it may be challenging to satisfy the assumptions underlying these techniques, e.g., that the verified device cannot communicate with a more powerful computer during the attestation process. Still, software- and hardware-based attestation mechanisms complement each other. Hardware-based attestation mechanisms can provide load-time guarantees, with software-based mechanisms helping to ensure tamper-evident execution. This reduces the threat posed by the TOCTOU vulnerability of the IMA.

6.2 Alternative User Interfaces

An important property achieved by the BitE system is that the user selects the registered application with which she would like to establish a trusted tunnel from a list on her trusted mobile device's screen. We present two alternative operating models that may be more convenient for the user, though they tradeoff the strength of the resulting trusted tunnel.

6.2.1 Active Selection

As described in Section 4, BitE requires the user to select an application from a list presented by the BitE Mobile Client when a registered application requests a trusted tunnel for input. An alternative structure, and one that may be preferable for legacy applications, is one where the user directs the BitE system to establish a trusted tunnel. One possibility for giving the user this ability is to maintain a list of all registered applications on the BitE Mobile Client. When the user wants to send secure input to, e.g., *OpenOffice Calc*, she selects "OpenOffice Calc" from the list on her mobile device.

This system has advantages for legacy applications since they do not actively request a trusted tunnel for input. BitE as described in Section 4.2 requires a legacy application to receive *all* input through the trusted tunnel, which may be inconvenient for the user if she wishes to interact with a second application with a trusted tunnel while her legacy application is still running (and using the trusted tunnel).

The drawback of this *Active Selection* scheme is that it requires user diligence. We are concerned about users forgetting to manually enable the trusted tunnel when they input sensitive data.

6.2.2 Always-On-Top

Another possible configuration for a trusted tunnel system is one where the window manager is involved. In this scenario, the BitE Mobile Client and the BitE Kernel Module maintain multiple active sessions. The user's typing goes to whatever application the window manager considers to be "on top." This configuration for BitE is problematic since the window manager becomes a part of the TCB. Much of the motivation for BitE is that it is able to remove the window manager from the TCB for trusted tunnel input.

7 Prototype and Evaluation

In this section, we describe our proof-of-concept prototype and evaluate some practical considerations necessary for actually building BitE. We have developed a J2ME MIDP 2.0 [19] BitE Mobile Client. It receives keystrokes via an infrared keyboard and sends them via Bluetooth to a prototype BitE Kernel Module loaded on the host platform. We use the BouncyCastle Lightweight Cryptography API for all cryptographic operations. Our prototype is shown in operation in Figure 4.

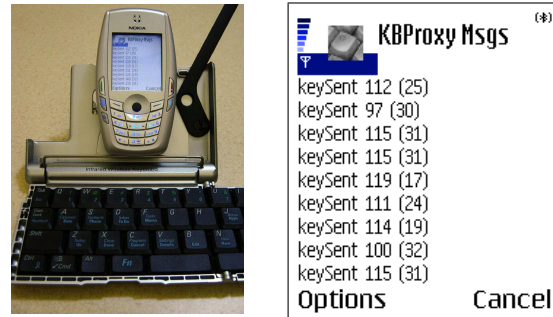


Figure 4: Our prototype BitE system and a debugging screenshot showing the prototype in operation.

The BitE Mobile Client consists of less than 1000 lines of Java code, not including source code from libraries. The BitE Kernel Module consists of approximately 500 lines of C and Java code which interacts with the BitE Mobile Client via Bluetooth, the legacy input system via the `uinput` kernel module, and the integrity measurement architecture of Sailer et al. [26] via the `/proc` filesystem.

7.1 Options for a Trusted Mobile Device

We decided to develop the BitE Mobile Client as a J2ME application because of the wide variety of mobile devices which support J2ME. In particular, millions of smartphones have already been sold which are capable of running the BitE Mobile Client. However, mobile phones are continuously increasing in complexity and thus feature software vulnerabilities of their own. A recent study places the number of existing worms and viruses for mobile phones at approximately 90 [15]. While 90 is a miniscule number when compared to the amount of malware in circulation which affects desktop computing platforms, it is still a significant figure when considering the amount of trust BitE puts in the mobile device. Efforts are ongoing to improve the security of mobile devices, augmented by the experience gained working to secure more traditional platforms [36]. Still, with today's technology, BitE is best run on a dedicated device whose software can only be upgraded under carefully controlled conditions.

7.2 Encrypted Channel Setup Latency Between Mobile Device and Host

We have performed experiments to determine the overhead associated with asymmetric cryptographic operations necessary to establish encrypted, authenticated communication between a mobile phone and host. We ran our J2ME MIDP 2.0 application on both a Nokia 6620 and a Nokia N70.

Establishing mutually authenticated communication involves performing asymmetric signature and verification operations at both communication endpoints. In our experiments with 1024-bit RSA keys (see Table 1), signing operations take 1757 ms and 1332 ms on average

Action	Nokia N70		Nokia 6620	
	Time (ms)	Variance	Time (ms)	Variance
RSA PSS (sign)	1332	171	1757	297
RSA verify	40	63	54	31
SHA-1 aggregate	91	125	171	110
Data manipulation	906	1000	2087	703

Table 1: Average time (in milliseconds) to perform an RSA signature and verification with a 1024-bit key and a public exponent of 65537; compute an aggregate hash of 401 (for the Nokia 6620) and 325 (for the Nokia N70) SHA-1 measurements from a Debian workstation running Linux kernel 2.6.12.5; and manipulate the IMA measurement list data.

for a Nokia 6620 and Nokia N70, respectively. Signature verification averages 54 ms and 40 ms, respectively. Thus, mutual authentication using either of these phones will take on the order of 3 to 4 seconds, which is a noticeable but tolerable delay. This is because these asymmetric operations are only required for communication setup. Once session keys are established, efficient symmetric primitives can be used for communication. The Nokia N70 consistently outperforms the Nokia 6620, but the margin is rather narrow.

7.3 Keyboard / Mobile Device Communication

We experimented with an infrared keyboard to provide user input to the BitE Mobile Client, and a Bluetooth connection from the BitE Mobile Client to the host platform. This is because our development phones can support only one Bluetooth connection at a time. The use of an infrared keyboard is undesirable because the keystrokes are transmitted in the clear. A better solution is to use a keyboard that physically attaches to the mobile phone. Alternatively, a Bluetooth keyboard capable of authenticated, encrypted communication can be used. We are unaware of any mobile phones available today that support more than one active Bluetooth connection simultaneously, but such devices may become available in the near future.

We performed simple usability experiments to analyze keystroke latencies, to ensure that BitE is not rendered useless by excessive latency while typing. We observe no noticeable latency with debug logging disabled. Use of symmetric cryptographic primitives introduces minimal overhead per keystroke. For example, the use of counter-mode encryption could enable the BitE Mobile Client to precompute enough of the key stream so that the only encrypt / decrypt operation on the critical path for a keystroke is an exclusive-or [17]. This reduces the most significant cryptographic per-keystroke operation to the verification of a MAC (e.g., HMAC-SHA-1, [3, 4, 14]). Our experiments below on verifying integrity measurements indicate that SHA-1 operations can be performed efficiently on the class of mobile phones we consider.

7.4 Verifying Attestations on the Mobile Device

We have the open-source integrity measurement architecture (IMA) of Sailer et al. [26] running on our development system. We have implemented the operations necessary for the BitE Kernel Module to send the IMA measurement list to the BitE Mobile Client, and for the BitE Mobile Client to compute the PCR aggregate for comparison with a signed PCR quote from the TPM. Table 1 shows some performance results for our prototype on a Nokia 6620 and N70. To validate an attested set of measurements from the host platform, the measurement list must be hashed for comparison with the signed PCR aggregate. For a typical desktop system, this involves hundreds of hash operations. Our experiments show that the average time necessary to compute a SHA-1 hash of 325 measurements (the number of measurements our development system had performed at the time of the experiment), 91 ms on the N70, is far less than the time necessary to manipulate the data from the measurement list—906 ms. In this experiment, the Nokia N70 is bound by memory access and not CPU operations. Our results show that the expected time for a Nokia N70 to verify an attestation (check the signature on the aggregate values from the PCRs, hash the measurement list, and compare the aggregate hash from the measurement list to the appropriate PCR value) is approximately 2 seconds, which will decrease with future devices. See Table 1 for the results from a corresponding experiment run on a Nokia 6620.

8 Conclusions

Bump in the Ether is a system that uses a trusted mobile device as a proxy between a keyboard and a TPM-equipped host platform to establish a trusted tunnel for user input to applications. The resulting tunnel is an end-to-end encrypted, authenticated tunnel all the way from a user's mobile device to an application running on the host platform. BitE places specific emphasis on these design issues: (1) malware such as software keyloggers, spyware, and Trojans running at user level will be unable to capture the user's input; (2) operation of BitE is convenient and intuitive for users; (3) BitE is feasible today on commodity hardware; and (4) BitE offers some protection for legacy applications.

9 Acknowledgements

We are grateful to Reiner Sailer and Leendert van Doorn for their assistance with the Integrity Measurement Architecture. We would also like to thank the anonymous reviewers for their valuable feedback.

This work was supported in part by National Science Foundation award number CCF-0424422.

References

- [1] D. Balfanz and E. W. Felten. Hand-held computers can be better smart cards. In *Proceedings of the USENIX Security Symposium*, August 1999.
- [2] D. Balfanz, D.K. Smetters, P. Stewart, and H. C. Wong. Talking to strangers: Authentication in ad-hoc wireless networks. In *Proceedings of the Symposium on Network and Distributed Systems Security*, February 2002.
- [3] M. Bellare, R. Canetti, and H. Krawczyk. Keyed hash functions and message authentication. In *Proceedings of Crypto*, pages 1–15, 1996.
- [4] M. Bellare, T. Kohno, and C. Namprempre. SSH transport layer encryption modes. Internet draft, August 2005.
- [5] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings. Compartmented mode workstation: Prototype highlights. *Software Engineering*, 16(6):608–618, June 1990.
- [6] M. Carson and J. Cugini. An X11-based Multilevel Window System architecture. In *Proceedings of the EUUG Technical Conference*, 1990.
- [7] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, August 2004.
- [8] J. Epstein. A prototype for Trusted X labeling policies. In *Proceedings of the Sixth Annual Computer Security Applications Conference*, December 1990.
- [9] J. Epstein and J. Picciotto. Issues in building Trusted X Window Systems. *The X Resource*, 1(1), Fall 1991.
- [10] J. Epstein and J. Picciotto. Trusting X: Issues in building Trusted X window systems -or- what's not trusted about X? In *Proceedings of the 14th Annual National Computer Security Conference*, October 1991.
- [11] G. Faden. Reconciling CMW requirements with those of X11 applications. In *Proceedings of the 14th Annual National Computer Security Conference*, October 1991.
- [12] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol: Version 3.0. Internet draft, Netscape Communications, November 1996.
- [13] T. Garfinkel, B. Pfaff, J. Chow, and M. Rosenblum. Data lifetime is a systems problem. In *Proceedings of the ACM SIGOPS European Workshop*, September 2004.
- [14] P. Jones. RFC 3174: US secure hash algorithm 1 (SHA1), September 2001.
- [15] N. Leavitt. Will proposed standard make mobile phones more secure? *IEEE Computer*, 38(12):20–22, 2005.
- [16] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-is-Believing: Using camera phones for human-verifiable authentication. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.
- [17] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press Series on Discrete Mathematics and its Applications. CRC Press, 1997.
- [18] Microsoft Corporation. Next generation secure computing base. <http://www.microsoft.com/resources/ngscb/>, April 2006.
- [19] Sun Microsystems. Mobile information device profile (MIDP) version 2.0, April 2006.
- [20] B. A. Myers. Using handhelds and PCs together. *Communications of the ACM*, 44(11), November 2001.
- [21] J. Picciotto. Towards trusted cut and paste in the X Window System. In *Proceedings of the Seventh Annual Computer Security Applications Conference*, December 1991.
- [22] J. Picciotto and J. Epstein. A comparison of Trusted X security policies, architectures, and interoperability. In *Proceedings of the Eighth Annual Computer Security Applications Conference*, December 1992.
- [23] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.
- [24] D. S. H. Rosenthal. LInX—a Less INsecure X server (Sun Microsystems unpublished draft).
- [25] S. J. Ross, J. L. Hill, M. Y. Chen, A. D. Joseph, D. E. Culler, and E. A. Brewer. A composable framework for secure multi-modal access to Internet services from post-PC devices. *Mobile Network Applications*, 7(5):389–406, 2002.
- [26] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the USENIX Security Symposium*, 2004.
- [27] J. Scott-Joynt. The enemy within. BBC News, Available at: <http://news.bbc.co.uk/2/hi/business/4510561.stm>, May 2005.
- [28] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles*, October 2005.
- [29] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proceedings of the USENIX Security Symposium*, 2004.
- [30] R. Sharp, A. Madhavapeddy, R. Want, T. Perring, and J. Light. Fighting crimeware: An architecture for split-trust web applications. Technical Report to appear, Intel Research Center, 2006.
- [31] R. Sharp, J. Scott, and A. Beresford. Secure mobile computing via public terminals. In *Proceedings of the International Conference on Pervasive Computing*, May 2006.
- [32] E. Shi, A. Perrig, and L. van Doorn. BIND: A time-of-use attestation service for secure distributed systems. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2005.
- [33] D. X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on SSH. In *Proceedings of the USENIX Security Symposium*, 2001.
- [34] F. Stajano and R. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Proceedings of the Security Protocols Workshop*, 1999.
- [35] J. T. Trostle. Timing attacks against trusted path. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1998.
- [36] Trusted Computing Group. Security in mobile phones whitepaper, October 2003.
- [37] Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands, October 2003. Version 1.2, Revision 62.
- [38] J. P. L. Woodward. Security requirements for system high and compartmented mode workstations. Technical Report MTR 9992, Rev. 1, The MITRE Corporation, November 1987.
- [39] The XFree86 project, Inc. <http://www.xfree86.org/>, April 2006.
- [40] The X.Org foundation. <http://www.x.org/>, April 2006.
- [41] L. Zhuang, F. Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. In *Proceedings of the ACM Conference on Computer and Communications Security*, October 2005.

Sharing Networked Resources with Brokered Leases

David Irwin, Jeffrey Chase, Laura Grit, Aydan Yumerefendi, and David Becker
Duke University

{irwin,chase,grit,aydan,becker}@cs.duke.edu

Kenneth G. Yocum
University of California, San Diego
kyocum@cs.ucsd.edu

Abstract

This paper presents the design and implementation of Shirako, a system for on-demand leasing of shared networked resources. Shirako is a prototype of a service-oriented architecture for resource providers and consumers to negotiate access to resources over time, arbitrated by brokers. It is based on a general lease abstraction: a lease represents a contract for some quantity of a typed resource over an interval of time. Resource types have attributes that define their performance behavior and degree of isolation.

Shirako decouples fundamental leasing mechanisms from resource allocation policies and the details of managing a specific resource or service. It offers an extensible interface for custom resource management policies and new resource types. We show how Shirako enables applications to lease groups of resources across multiple autonomous sites, adapt to the dynamics of resource competition and changing load, and guide configuration and deployment. Experiments with the prototype quantify the costs and scalability of the leasing mechanisms, and the impact of lease terms on fidelity and adaptation.

1 Introduction

Managing shared cyberinfrastructure resources is a fundamental challenge for service hosting and utility computing environments, as well as the next generation of network testbeds and grids. This paper investigates an approach to networked resource sharing based on the foundational abstraction of *resource leasing*.

We present the design and implementation of Shirako, a toolkit for a brokered utility service architecture.¹ Shirako is based on a common, extensible resource leasing abstraction that can meet the evolving needs of several strains of systems for networked resource sharing—whether the resources are held in common by a commu-

nity of shareholders, offered as a commercial hosting service to paying customers, or contributed in a reciprocal fashion by self-interested peers. The Shirako architecture reflects several objectives:

- *Autonomous providers.* A provider is any administrative authority that controls resources; we refer to providers as *sites*. Sites may contribute resources to the system on a temporary basis, and retain ultimate control over their resources.
- *Adaptive guest applications.* The clients of the leasing services are hosted application environments and managers acting on their behalf. We refer to these as *guests*. Guests use programmatic lease service interfaces to acquire resources, monitor their status, and adapt to the dynamics of resource competition or changing demand (e.g., flash crowds).
- *Pluggable resource types.* The leased infrastructure includes edge resources such as servers and storage, and may also include resources within the network itself. Both the owning site and the guest supply type-specific configuration actions for each resource; these execute in sequence to setup or tear down resources for use by the guest, guided by configuration properties specified by both parties.
- *Brokering.* Sites delegate limited power to allocate their resource offerings—possibly on a temporary basis—by registering their offerings with one or more brokers. Brokers export a service interface for guests to acquire resources of multiple types and from multiple providers.
- *Extensible allocation policies.* The dynamic assignment of resources to guests emerges from the interaction of policies in the guests, sites, and brokers. Shirako defines interfaces for resource policy modules at each of the policy decision points.

Section 2 gives an overview of the Shirako leasing services, and an example site manager for on-demand cluster sites. Section 3 describes the key elements of the system design: generic property sets to describe resources and guide their configuration, scriptable configuration ac-

¹This research is supported by the National Science Foundation through ANI-0330658 and CNS-0509408, and by IBM, HP Labs, and Network Appliance. Laura Grit is a National Physical Science Consortium Fellow.

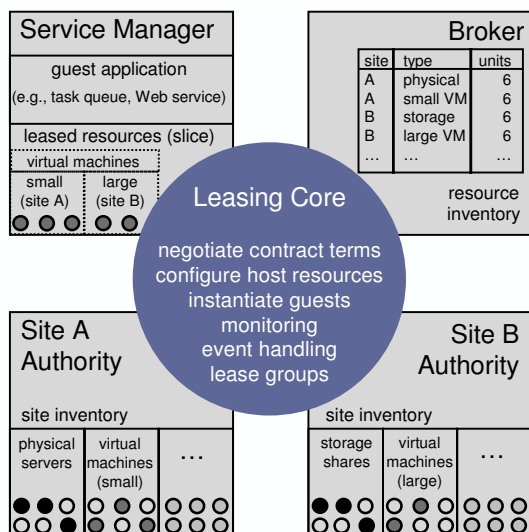


Figure 1: An example scenario with a guest application acquiring resources from two cluster sites through a broker. Each resource provider site has a server (site authority) that controls its resources, and registers inventories of offered resources with the broker. A service manager negotiates with the broker and authorities for leases on behalf of the guest. A common lease package manages the protocol interactions and lease state for all actors. The Shirako leasing core is resource-independent, application-independent, and policy-neutral.

tions, support for lease extends with resource flexing, and abstractions for grouping related leases. Section 4 summarizes the implementation, and Section 5 presents experimental results from the prototype. The experiments evaluate the overhead of the leasing mechanisms and the use of leases to adapt to changes in demand. Section 6 sets Shirako in context with related work.

2 Overview

Shirako's leasing architecture derives from the SHARP framework for secure resource peering and distributed resource allocation [13]. The participants in the leasing protocols are long-lived software entities (*actors*) that interact over a network to manage resources.

- Each guest has an associated *service manager* that monitors application demands and resource status, and negotiates to acquire leases for the mix of resources needed to host the guest. Each service manager requests and maintains leases on behalf of one or more guests, driven by its own knowledge of application behavior and demand.
- An *authority* controls resource allocation at each resource provider site or domain, and is responsible for enforcing isolation among multiple guests hosted on the resources under its control.
- *Brokers* (agents) maintain inventories of resources offered by sites, and match requests with their re-

source supply. A site may maintain its own broker to keep control of its resources, or delegate partial, temporary control to third-party brokers that aggregate resource inventories from multiple sites.

These actors may represent different trust domains and identities, and may enter into various trust relationships or contracts with other actors.

2.1 Cluster Sites

One goal of this paper is to show how dynamic, brokered leasing is a foundation for resource sharing in networked clusters. For this purpose we introduce a cluster site manager to serve as a running example. The system is an implementation of Cluster-On-Demand (COD [7]), rearchitected as an authority-side Shirako plugin.

The COD site authority exports a service to allocate and configure *virtual clusters* from a shared server cluster. Each virtual cluster comprises a dynamic set of nodes and associated resources assigned to some guest at the site. COD provides basic services for booting and imaging, naming and addressing, and binding storage volumes and user accounts on a per-guest basis. In our experiments the leased virtual clusters have an assurance of performance isolation: the nodes are either physical servers or Xen [2] virtual machines with assigned shares of node resources.

Figure 1 depicts an example of a guest service manager leasing a distributed cluster from two COD sites. The site authorities control their resources and configure the virtual clusters, in this case by instantiating nodes running a guest-selected image. The service manager deploys and monitors the guest environment on the nodes. The guest in this example may be a distributed service or application, or a networked environment that further subdivides the resources assigned to it, e.g., a cross-institutional grid or content distribution network.

The COD project began in 2001 as an outgrowth of our work on dynamic resource provisioning in hosting centers [6]. Previous work [7] describes an earlier COD prototype, which had an ad hoc leasing model with built-in resource dependencies, a weak separation of policy and mechanism, and no ability to delegate or extend provisioning policy or to coordinate resource usage across federated sites. Our experience with COD led us to pursue a more general lease abstraction with distributed, accountable control in SHARP [13], which was initially prototyped for PlanetLab [4]. We believe that dynamic leasing is a useful basis to coordinate resource sharing for other systems that create distributed virtual execution environments from networked virtual machines [9, 17, 18, 19, 20, 25, 26, 28].

2.2 Resource Leases

The resources leased to a guest may span multiple sites and may include a diversity of resource types in differing quantities. Each SHARP resource has a *type* with associ-

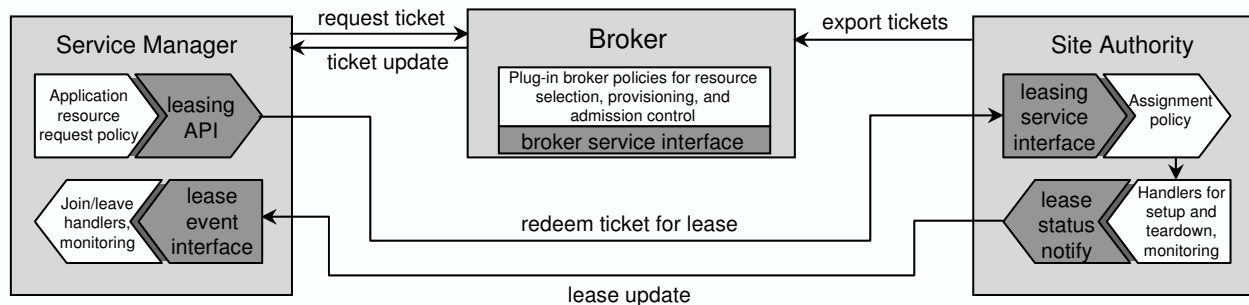


Figure 2: Summary of protocol interactions and extension points for the leasing system. An application-specific service manager uses the lease API to request resources from a broker. The broker issues a ticket for a resource type, quantity, and site location that matches the request. The service manager requests a lease from the owning site authority, which selects the resource units, configures them (*setup*), and returns a lease to the service manager. The arriving lease triggers a *join* event for each resource unit joining the guest; the join handler installs the new resources into the application. Plug-in modules include the broker provisioning policy, the authority assignment policy, and the *setup* and *join* event handlers.

ated attributes that characterize the function and power of instances or *units* of that type. Resource units with the same type at a site are presumed to be interchangeable.

Each lease binds a set of resource units from a site (a *resource set*) to a guest for some time interval (*term*). A lease is a contract between a site and a service manager: the site makes the resources available to the guest identity for the duration of the lease term, and the guest assumes responsibility for any use of the resources by its identity. In our current implementation each lease represents some number of units of resources of a single type.

Resource attributes define the performance and predictability that a lease holder can expect from the resources. Our intent is that the resource attributes quantify capability in an application-independent way. For example, a lease could represent a *reservation* for a block of machines with specified processor and memory attributes (clock speed etc.), or a storage partition represented by attributes such as capacity, spindle count, seek time, and transfer speed. Alternatively, the resource attributes could specify a weak assurance, such as a best-effort service contract or probabilistically overbooked shares.

2.3 Brokers

Guests with diverse needs may wish to acquire and manage multiple leases in a coordinated way. In particular, a guest may choose to aggregate resources from multiple sites for geographic dispersion or to select preferred suppliers in a competitive market.

Brokers play a key role because they can coordinate resource allocation across sites. SHARP brokers are responsible for *provisioning*: they determine *how much* of each resource type each guest will receive, and *when*, and *where*. The sites control how much of their inventory is offered for leasing, and by which brokers, and when. The site authorities also control the *assignment* of specific re-

source units at the site to satisfy requests approved by the brokers. This decoupling balances global coordination (in the brokers) with local autonomy (in the site authorities).

Figure 2 depicts a broker's role as an intermediary to arbitrate resource requests. The broker approves a request for resources by issuing a *ticket* that is redeemable for a lease at some authority, subject to certain checks at the authority. The ticket specifies the resource type and the number of units granted, and the interval over which the ticket is valid (the term). Sites issue tickets for their resources to the brokers; the broker arbitration policy may subdivide any valid ticket held by the broker. All SHARP exchanges are digitally signed, and the broker endorses the public keys of the service manager and site authority. Previous work presents the SHARP delegation and security model in more detail, and mechanisms for accountable resource contracts [13].

2.4 System Goals

Shirako is a toolkit for constructing service managers, brokers, and authorities, based on a common, extensible leasing core. A key design principle is to factor out any dependencies on resources, applications, or resource management policies from the core. This decoupling serves several goals:

- The resource model should be sufficiently general for other resources such as bandwidth-provisioned network paths, network storage objects, or sensors. It should be possible to allocate and configure diverse resources alone or in combination.
- Shirako should support development of guest applications that adapt to changing conditions. For example, a guest may respond to load surges or resource failures by leasing additional resources, or it may adjust to contention for shared resources by deferring work or reducing service quality. Resource sharing

expands both the need and the opportunity for adaptation.

- Shirako should make it easy to deploy a range of approaches and policies for resource allocation in the brokers and sites. For example, Shirako could serve as a foundation for a future resource economy involving bidding, auctions, futures reservations, and combinatorial aggregation of resource bundles. The software should also run in an emulation mode, to enable realistic experiments at scales beyond the available dedicated infrastructure.

Note that Shirako has no globally trusted core; rather, one contribution of the architecture is a clear factoring of powers and responsibilities across a dynamic collection of participating actors, and across pluggable policy modules and resource drivers within the actor implementations.

3 Design

Shirako comprises a generic leasing core with plug-in interfaces for extension modules for policies and resource types. The core manages state storage and recovery for the actors, and mediates their protocol interactions. Each actor may invoke primitives in the core to initiate lease-related actions at a time of its choosing. In addition, actor implementations supply plug-in extension modules that are invoked from the core in response to specific events. Most such events are associated with resources transferring in or out of a *slice*—a logical grouping for resources held by a given guest.

Figure 2 summarizes the separation of the core from the plugins. Each actor has a *mapper* policy module that is invoked periodically, driven by a clock. On the service manager, the mapper determines when and how to redeem existing tickets, extend existing leases, or acquire new leases to meet changing demand. On the broker and authority servers, the mappers match accumulated pending requests with resources under the server's control. The broker mapper deals with resource provisioning: it prioritizes ticket requests and selects resource types and quantities to fill them. The authority mapper assigns specific resource units from its inventory to fill lease requests that are backed by a valid ticket from an approved broker.

Service managers and authorities register *resource driver* modules defining resource-specific configuration actions. In particular, each resource driver has a pair of event handlers that drive configuration and membership transitions in the guest as resource units transfer in or out of a slice.

- The authority invokes a *setup* action to configure (*prime*) each new resource unit assigned to a slice by the mapper. The authority issues the lease when all of its setup actions have completed.
- The service manager invokes a *join* action to notify the guest of each new resource unit. Join actions are

driven by arriving lease updates.

- *Leave* and *teardown* actions close down resource units at the guest and site respectively. These actions are triggered by a lease expiration or resource failure.

3.1 Properties

Shirako actors must exchange context-specific information to guide the policies and configuration actions. For example, a guest expresses the resources requested for a ticket, and it may have specific requirements for configuring those resources at the site. It is difficult to maintain a clean decoupling, because this resource-specific or guest-specific information passes through the core.

Shirako represents all such context-specific information in *property lists* attached as attributes in requests, tickets, and leases. The property lists are sets of (*key*, *value*) string pairs that are opaque to the core; their meaning is a convention among the plugins. Property sets flow from one actor to another and through the plugins on each of the steps and protocol exchanges depicted in Figure 2.

- *Request properties* specify desired attributes and/or value for resources requested from a broker.
- *Resource properties* attached to tickets give the attributes of the assigned resource types.
- *Configuration properties* attached to redeem requests direct how the resources are to be configured.
- *Unit properties* attached to each lease define additional attributes for each resource unit assigned.

3.2 Broker Requests

The Shirako prototype includes a basic broker mapper with several important features driven by request properties. For example, a service manager may set request properties to define a range of acceptable outcomes.

- Marking a request as *elastic* informs the broker that the guest will accept fewer resource units if the broker is unable to fill its entire request.
- Marking a request as *defferable* informs the broker that the guest will accept a later start time if its requested start time is unavailable; for example, a service manager may request resources for an experiment, then launch the experiment automatically when the resources are available.

Request properties may also express additional constraints on a request. For example, the guest may mark a set of ticket requests as members of a *request group*, indicating that the broker must fill the requests atomically, with the same terms. The service manager tags one of its lease requests as the group leader, specifying a unique *groupID* and a *leaseCount* property giving the number of requests in the group. Each request has a *groupID* property identifying its request group, if any.

Resource type properties: passed from broker to service manager		
machine.memory	Amount of memory for nodes of this type	2GB
machine.cpu	CPU identifying string for nodes of this type	Intel Pentium4
machine.clockspeed	CPU clock speed for nodes of this type	3.2 GHz
machine.cpus	Number of CPUs for nodes of this type	2
Configuration properties: passed from service manager to authority		
image.id	Unique identifier for an OS kernel image selected by the guest and approved by the site authority for booting	Debian Linux
subnet.name	Subnet name for this virtual cluster	cats
host.prefix	Hostname prefix to use for nodes from this lease	cats
host.visible	Assign a public IP address to nodes from this lease?	true
admin.key	Public key authorized by the guest for root/admin access for nodes from this lease	[binary encoded]
Unit properties: passed from authority to service manager		
host.name	Hostname assigned to this node	cats01.cats.cod.duke.edu
host.privIPAddr	Private IP address for this node	172.16.64.8
host.pubIPAddr	Public IP address for this node (if any)	152.3.140.22
host.key	Host public key to authenticate this host for SSL/SSH	[binary encoded]
subnet.privNetmask	Private subnet mask for this virtual cluster	255.255.255.0

Table 1: Selected properties used by Cluster-on-Demand, and sample values.

When all leases for a group have arrived, the broker schedules them for a common start time when it can satisfy the entire group request. Because request groups are implemented within a broker—and because SHARP brokers have allocation power—a co-scheduled request group can encompass a variety of resource types across multiple sites. The default broker requires that request groups are always `defferable` and never `elastic`, so a simple FCFS scheduling algorithm is sufficient.

The request properties may also guide resource selection and arbitration under constraint. For example, we use them to encode bids for economic resource management [16]. They also enable attribute-based resource selection of types to satisfy a given request. A number of projects have investigated the matching problem, most recently in SWORD [22].

3.3 Configuring Virtual Clusters

The COD plugins use the configuration and unit properties to drive virtual cluster configuration (at the site) and application deployment (in the guest). Table 1 lists some important properties used in COD. These property names and legal values are conventions among the package classes for COD service managers and authorities.

To represent the wide range of actions that may be needed, the COD resource driver event handlers are scripted using *Ant* [1], an open-source OS-independent XML scripting package. Ant scripts invoke a library of packaged tasks to execute commands remotely and to manage network elements and application components. Ant is in wide use, and new plug-in tasks continue to become available. A Shirako actor may load XML Ant scripts dynamically from user-specified files, and actors may exchange Ant scripts across the network and execute them directly. When an event handler triggers, Ant substitutes variables within the script with the values of named

properties associated with the node, its containing lease, and its containing slice.

The *setup* and *teardown* event handlers execute within the site's trusted computing base (TCB). A COD site authority controls physical boot services, and it is empowered to run commands within the control domain on servers installed with a Xen hypervisor, to create new virtual machines or change the resources assigned to a virtual machine. The site operator must approve any authority-side resource driver scripts, although it could configure the actor to accept new scripts from a trusted repository or service manager.

Several configuration properties allow a COD service manager to guide authority-side configuration.

- *OS boot image selection.* The service manager passes a string to identify an OS configuration from among a menu of options approved by the site authority as compatible with the machine type.
- *IP addressing.* The site assigns public IP addresses to nodes if the `visible` property is set.
- *Secure node access.* The site and guest exchange keys to enable secure, programmatic access to the leased nodes using SSL/SSH. The service manager generates a keypair and passes the public key as a configuration property. The site's *setup* handler writes the public key and a locally generated host private key onto the node image, and returns the host public key as a unit property.

The *join* and *leave* handlers execute outside of the site authority's TCB; they operate within the isolation boundaries that the authority has established for the slice and its resources. The unit properties returned for each node include the names and keys to allow the *join* handler to connect to the node to initiate *post-install* actions. In our prototype, a service manager is empowered to connect with root access and install arbitrary application soft-

ware. The *join* and *leave* event handlers also interact with other application components to reconfigure the application for membership changes. For example, the handlers could link to standard entry points of a Group Membership Service that maintains a consistent view of membership across a distributed application.

Ant has a sizable library of packaged tasks to build, configure, deploy, and launch software packages on various operating systems and Web application servers. The COD prototype includes service manager scripts to launch applications directly on leased resources, launch and dynamically resize cluster job schedulers (SGE and PBS), instantiate and/or automount NFS file volumes, and load Web applications within a virtual cluster.

3.4 Extend and Flex

There is a continuum of alternatives for adaptive resource allocation with leases. The most flexible model would permit actors to renegotiate lease contracts at any time. At the other extreme, a restrictive model might disallow any changes to a contract once it is made. Shirako leases may be extended (renewed) by mutual agreement. Peers may negotiate limited changes to the lease at renewal time, including *flexing* the number of resource units. In our prototype, changes to a renewed lease take effect only at the end of its previously agreed term.

The protocol to extend a lease involves the same pattern of exchanges as to initiate a new lease (see Figure 2). The service manager must obtain a new ticket from the broker; the ticket is marked as extending an existing ticket named by a unique ID. Renewals maintain the continuity of resource assignments when both parties agree to extend the original contract. An extend makes explicit that the next holder of a resource is the same as the current holder, bypassing the usual *teardown/setup* sequence at term boundaries. Extends also free the holder from the risk of a forced migration to a new resource assignment—assuming the renew request is honored.

With support for resource flexing, a guest can obtain these benefits even under changing demand. Without flex extends, a guest with growing resource demands is forced to instantiate a new lease for the residual demand, leading to a fragmentation of resources across a larger number of leases. Shrinking a slice would force a service manager to vacate a lease and replace it with a smaller one, interrupting continuity.

Flex extends turned out to be a significant source of complexity. For example, resource assignment on the authority must be sequenced with care to process shrinking extends first, then growing extends, then new redeems. One drawback of our current system is that a Shirako service manager has no general way to name victim units to relinquish on a shrinking extend; COD overloads configuration properties to cover this need.

Common lease core	2755
Actor state machines	1337
Cluster-on-Demand	3450
Policy modules (mappers)	1941
Calendar support for mappers	1179
Utility classes	1298

Table 2: Lines of Java code for Shirako/COD.

3.5 Lease Groups

Our initial experience with SHARP and Shirako convinced us that associating leases in *lease groups* as an important requirement. Section 3.2 outlines the related concept of *request groups*, in which a broker co-schedules grouped requests. Also, since the guest specifies properties on a per-lease basis, it is useful to obtain separate leases to allow diversity of resources and their configuration. Configuration dependencies among leases may impose a partial order on configuration actions—either within the authority (*setup*) or within the service manager (*join*), or both. For example, consider a batch task service with a master server, worker nodes, and a file server obtained with separate leases: the file server must initialize before the master can *setup*, and the master must activate before the workers can *join* the service.

The Shirako leasing core enforces a specified configuration sequencing for lease groups on the service manager. It represents dependencies as a restricted form of DAG: each lease has at most one *redeem predecessor* and at most one *join predecessor*. If there is a redeem predecessor and the service manager has not yet received a lease for it, then it transitions the ticketed request into a blocked state, and does not redeem the ticket until the predecessor lease arrives, indicating that its *setup* is complete. Also, if a join predecessor exists, the service manager holds the lease in a blocked state and does not fire its *join* until the join predecessor is active. In both cases, the core upcalls a plugin method before transitioning out of the blocked state; the upcall gives the plugin an opportunity to manipulate properties on the lease before it fires, or to impose more complex trigger conditions.

4 Implementation

A Shirako deployment runs as a dynamic collection of interacting peers that work together to coordinate asynchronous actions on the underlying resources. Each actor is a multithreaded server written in Java and running within a Java Virtual Machine. Actors communicate using an asynchronous peer-to-peer messaging model through a replaceable stub layer. SOAP stubs allow actors running in different JVMs to interact using Web Services protocols (Apache Axis).

Our goal was to build a common toolkit for all actors that is understandable and maintainable by one person. Table 2 shows the number of lines of Java code (semi-

colon lines) in the major system components of our prototype. In addition, there is a smaller body of code, definitions, and stubs to instantiate groups of Shirako actors from XML descriptors, encode and decode actor exchanges using SOAP messaging, and sign and validate SHARP-compliant exchanges. Shirako also includes a few dozen Ant scripts, averaging about 40 lines each, and other supporting scripts. These scripts configure the various resources and applications that we have experimented with, including those described in Section 5. Finally, the system includes a basic Web interface for Shirako/COD actors; it is implemented in about 2400 lines of Velocity scripting code that invokes Java methods directly.

The prototype makes use of several other open-source components. It uses Java-based tools to interact with resources when possible, in part because Java exception handling is a basis for error detection, reporting, attribution, and logging of configuration actions. Ant tasks and the Ant interpreter are written in Java, so the COD resource drivers execute configuration scripts by invoking the Ant interpreter directly within the same JVM. The event handlers often connect to nodes using key-based logins through *jsch*, a Java secure channel interface (SSH2). Actors optionally use *jldap* to interface to external LDAP repositories for recovery. COD employs several open-source components for network management based on LDAP directory servers (RFC 2307 schema standard) as described below.

4.1 Lease State Machines

The Shirako core must accommodate long-running asynchronous operations on lease objects. For example, the brokers may delay or batch requests arbitrarily, and the *setup* and *join* event handlers may take seconds, minutes, or hours to configure resources or integrate them into a guest environment. A key design choice was to structure the core as a non-blocking event-based state machine from the outset, rather than representing the state of pending operations on the stacks of threads, e.g., blocked in RPC calls. The lease state represents any pending action until a completion event triggers a state transition. Each of the three actor roles has a separate state machine.

Figure 3 illustrates typical state transitions for a resource lease through time. The state for a brokered lease spans three interacting state machines, one in each of the three principal actors involved in the lease: the service manager that requests the resources, the broker that provisions them, and the authority that owns and assigns them. Thus the complete state space for a lease is the cross-product of the state spaces for the actor state machines. The state combinations total about 360, of which about 30 are legal and reachable.

The lease state machines govern all functions of the core leasing package. State transitions in each actor are initiated by arriving requests or lease/ticket updates, and

by events such as the passage of time or changes in resource status. Actions associated with each transition may invoke a plugin, commit modified lease state and properties to an external repository, and/or generate a message to another actor. The service manager state machine is the most complex because the brokering architecture requires it to maintain ticket status and lease status independently. For example, the *ActiveTicketed* state means that the lease is active and has obtained a ticket to renew, but it has not yet redeemed the ticket to complete the lease extension. The broker and authority state machines are independent; in fact, the authority and broker interact only when resource rights are initially delegated to the broker.

The concurrency architecture promotes a clean separation of the leasing core from resource-specific code. The resource handlers—*setup/teardown*, *join/leave*, and status *probe* calls—do not hold locks on the state machines or update lease states directly. This constraint leaves them free to manage their own concurrency, e.g., by using blocking threads internally. For example, the COD node drivers start a thread to execute a designated target in an Ant script. In general, state machine threads block only when writing lease state to a repository after transitions, so servers need only a small number of threads to provide sufficient concurrency.

4.2 Time and Emulation

Some state transitions are triggered by timer events, since leases activate and expire at specified times. For instance, a service manager may schedule to shutdown a service on a resource before the end of the lease. Because of the importance of time in the lease management, actor clocks should be loosely synchronized using a time service such as NTP. While the state machines are robust to timing errors, unsynchronized clocks can lead to anomalies from the perspective of one or more actors: requests for leases at a given start time may be rejected because they arrive too late, or they may activate later than expected, or expire earlier than expected. One drawback of leases is that managers may “cheat” by manipulating their clocks; accountable clock synchronization is an open problem.

When control of a resource passes from one lease to another, we charge setup time to the controlling lease, and teardown time to the successor. Each holder is compensated fairly for the charge because it does not pay its own teardown costs, and teardown delays are bounded. This design choice greatly simplifies policy: brokers may allocate each resource to contiguous lease terms, with no need to “mind the gap” and account for transfer costs. Similarly, service managers are free to vacate their leases just before expiration without concern for the authority-side teardown time. Of course, each guest is still responsible for completing its *leave* operations before the lease expires: the authority is empowered to unilaterally initiate *teardown* whether the guest is ready or not.

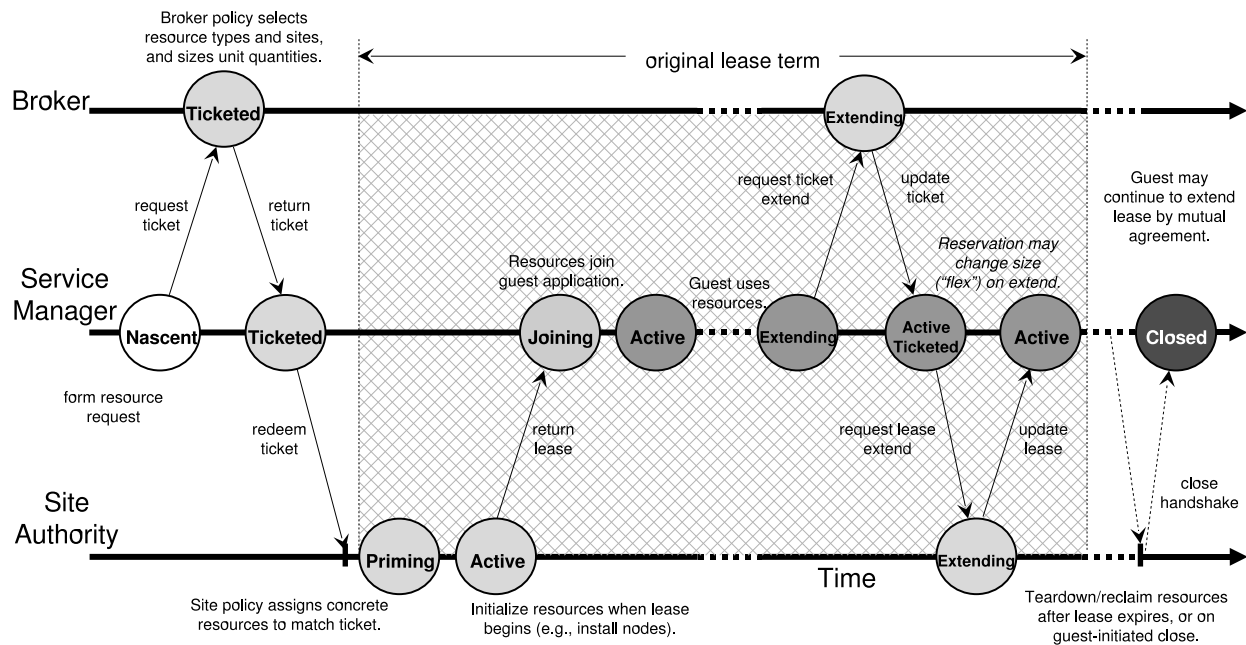


Figure 3: Interacting lease state machines across three actors. A lease progresses through an ordered sequence of states until it is active; the rate of progress may be limited by delays imposed in the policy modules or by latencies to configure resources. Failures lead to retries or to error states reported back to the service manager. Once the lease is active, the service manager may initiate transitions through a cycle of states to extend the lease. Termination involves a handshake similar to TCP connection shutdown.

Actors are externally clocked to eliminate any dependency on absolute time. Time-related state transitions are driven by a *virtual clock* that advances in response to external *tick* calls. This feature is useful to exercise the system and control the timing and order of events. In particular, it enables emulation experiments in virtual time, as for several of the experiments in Section 5. The emulations run with null resource drivers that impose various delays but do not actually interact with external resources. All actors retain and cache lease state in memory, in part to enable lightweight emulation-mode experiments without an external repository.

4.3 Cluster Management

COD was initially designed to control physical machines with database-driven network booting (PXE/DHCP). The physical booting machinery is familiar from Emulab [28], Rocks [23], and recent commercial systems. In addition to controlling the IP address bindings assigned by PXE/DHCP, the node driver controls boot images and options by generating configuration files served via TFTP to standard bootloaders (e.g., *grub*).

A COD site authority drives cluster reconfiguration in part by writing to an external directory server. The COD schema is a superset of the RFC 2307 standard schema for a Network Information Service based on LDAP directories. Standard open-source services exist to administer networks from a LDAP repository compliant with RFC

2307. The DNS server for the site is an LDAP-enabled version of BIND9, and for physical booting we use an LDAP-enabled DHCP server from the Internet Systems Consortium (ISC). In addition, guest nodes have read access to an LDAP directory describing the containing virtual cluster. Guest nodes configured to run Linux use an LDAP-enabled version of AutoFS to mount NFS file systems, and a PAM/NSS module that retrieves user logins from LDAP.

COD should be comfortable for cluster site operators to adopt, especially if they already use RFC 2307/LDAP for administration. The directory server is authoritative: if the COD site authority fails, the disposition of the cluster is unaffected until it recovers. Operators may override the COD server with tools that access the LDAP configuration directory.

4.4 COD and Xen

In addition to the node drivers, COD includes classes to manage node sets and IP and DNS name spaces at the slice level. The authority names each instantiated node with an ID that is unique within the slice. It derives node hostnames from the ID and a specified prefix, and allocates private IP addresses as offsets in a subnet block reserved for the virtual cluster when the first node is assigned to it. Although public address space is limited, our prototype does not yet treat it as a managed resource. In our deployment the service managers run on a control

subnet with routes to and from the private IP subnets.

In a further test of the Shirako architecture, we extended COD to manage virtual machines using the Xen hypervisor [2]. The extensions consist primarily of a modified node driver plugin and extensions to the authority-side mapper policy module to assign virtual machine images to physical machines. The new virtual node driver controls booting by opening a secure connection to the privileged control domain on the Xen node, and issuing commands to instantiate and control Xen virtual machines. Only a few hundred lines of code know the difference between physical and virtual machines. The combination of support for both physical and virtual machines offers useful flexibility: it is possible to assign blocks of physical machines dynamically to boot Xen, then add them to a resource pool to host new virtual machines.

COD install actions for node *setup* include some or all of the following: writing LDAP records; generating a bootloader configuration for a physical node, or instantiating a virtual machine; staging and preparing the OS image, running in the Xen control domain or on an OS-dependent trampoline such as Knoppix on the physical node; and initiating the boot. The authority writes some configuration-specific data onto the image, including the admin public keys and host private key, and an LDAP path reference for the containing virtual cluster.

5 Experimental Results

We evaluate the Shirako/COD prototype under emulation and in a real deployment. All experiments run on a testbed of IBM x335 rackmount servers, each with a single 2.8Ghz Intel Xeon processor and 1GB of memory. Some servers run Xen's virtual machine monitor version 3.0 to create virtual machines. All experiments run using Sun's Java Virtual Machine (JVM) version 1.4.2. COD uses OpenLDAP version 2.2.23-8, ISC's DHCP version 3.0.1rc11, and TFTP version 0.40-4.1 to drive network boots. Service manager, broker, and site authority Web Services use Apache Axis 1.2RC2.

Most experiments run all actors on one physical server within a single JVM. The actors interact through local proxy stubs that substitute local method calls for network communication, and copy all arguments and responses. When LDAP is used, all actors are served by a single LDAP server on the same LAN segment. Note that these choices are conservative in that the management overhead concentrates on a single server. Section 5.3 gives results using SOAP/XML messaging among the actors.

5.1 Application Performance

We first examine the latency and overhead to lease a virtual cluster for a sample guest application, the CardioWave parallel MPI heart simulator [24]. A service manager requests two leases: one for a coordinator node to launch the MPI job and another for a variable-sized

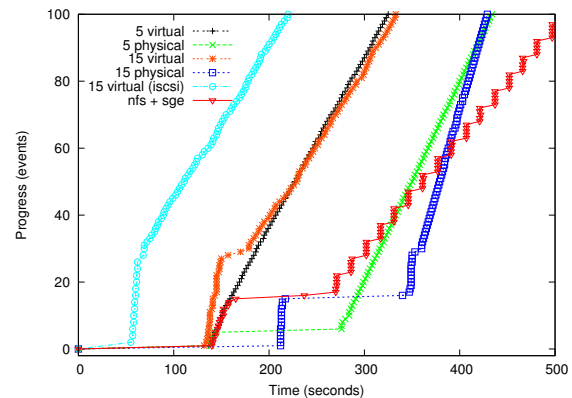


Figure 4: The progress of *setup* and *join* events and CardioWave execution on leased virtual clusters. The slope of each line gives the rate of progress. Xen clusters (left) activate faster and more reliably, but run slower than leased physical nodes (right). The step line shows an SGE batch scheduling service instantiated and subjected to a synthetic load. The fastest boot times are for VMs with flash-cloned iSCSI roots (far left).

block of worker nodes to run the job. It groups and sequences the lease joins as described in Section 3.5 so that all workers activate before the coordinator. The *join* handler launches CardioWave programmatically when the virtual cluster is fully active.

Figure 4 charts the progress of lease activation and the CardioWave run for virtual clusters of 5 and 15 nodes, using both physical and Xen virtual machines, all with 512MB of available memory. The guest earns progress points for each completed node join and each block of completed iterations in CardioWave. Each line shows: (1) an initial flat portion as the authority prepares a file system image for each node and initiates boots; (2) a step up as nodes boot and join, (3) a second flatter portion indicating some straggling nodes, and (4) a linear segment that tracks the rate at which the application completes useful work on the virtual cluster once it is running.

The authority prepares each node image by loading a 210MB compressed image (Debian Linux 2.4.25) from a shared file server and writing the 534MB uncompressed image on a local disk partition. Some node setup delays result from contention to load the images from a shared NFS server, demonstrating the value of smarter image distribution (e.g., [15]). The left-most line in Figure 4 also shows the results of an experiment with iSCSI root drives flash-cloned by the *setup* script from a Network Appliance FAS3020 filer. Cloning iSCSI roots reduces VM configuration time to approximately 35 seconds. Network booting of physical nodes is slower than Xen and shows higher variability across servers, indicating instability in the platform, bootloader, or boot services.

Cardiowave is an I/O-intensive MPI application. It shows better scaling on physical nodes, but its perfor-

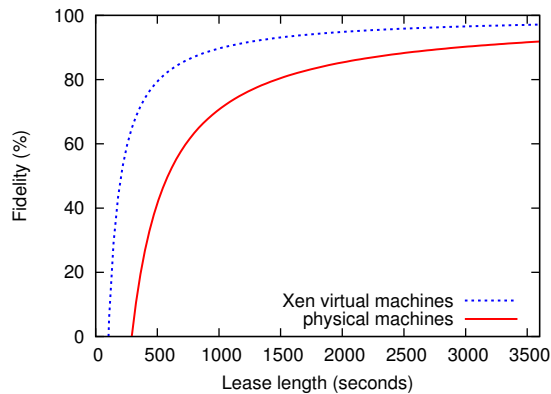


Figure 5: Fidelity is the percentage of the lease term usable by the guest application, excluding setup costs. Xen VMs are faster to *setup* than physical machines, yielding better fidelity.

mance degrades beyond ten nodes. With five nodes the Xen cluster is 14% slower than the physical cluster, and with 15 nodes it is 37% slower. For a long CardioWave run, the added Xen VM overhead outweighs the higher setup cost to lease physical nodes.

A more typical usage of COD in this setting would be to instantiate batch task services on virtual compute clusters [7], and let them schedule CardioWave and other jobs without rebooting the nodes. Figure 4 includes a line showing the time to instantiate a leased virtual cluster comprising five Xen nodes and an NFS file server, launch a standard Sun GridEngine (SGE) job scheduling service on it, and subject it to a synthetic task load. This example uses lease groups to sequence configuration as described in Section 3.5. The service manager also stages a small data set (about 200 MB) to the NFS server, increasing the activation time. The steps in the line correspond to simultaneous completion of synthetic tasks on the workers.

Figure 5 uses the *setup/join/leave/teardown* costs from the previous experiment to estimate their effect on the system's *fidelity* to its lease contracts. Fidelity is the percentage of the lease term that the guest application is able to use its resources. Amortizing these costs over longer lease terms improves fidelity. Since physical machines take longer to *setup* than Xen virtual machines, they have a lower fidelity and require longer leases to amortize their costs.

5.2 Adaptivity to Changing Load

This section demonstrates the role of brokers to arbitrate resources under changing workload, and coordinate resource allocation from multiple sites. This experiment runs under emulation (as described in Section 4.2) with null resource drivers, virtual time, and lease state stored only in memory (no LDAP). In all other respects the emulations are identical to a real deployment. We use two emulated 70-node cluster sites with a shared broker. The

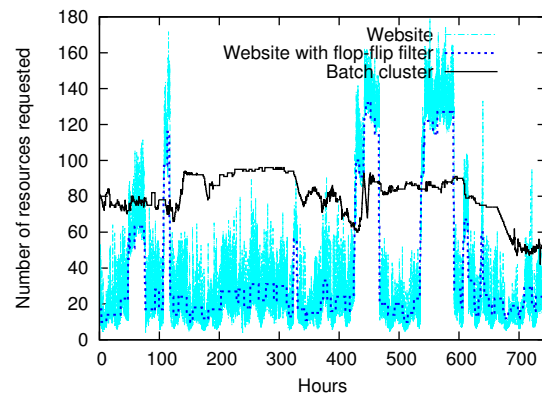
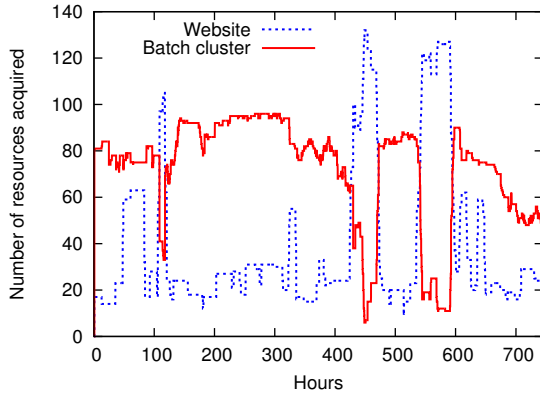


Figure 6: Scaled resource demands for one-month traces from an e-commerce website and a production batch cluster. The e-commerce load signal is smoothed with a *flop-flip* filter for stable dynamic provisioning.

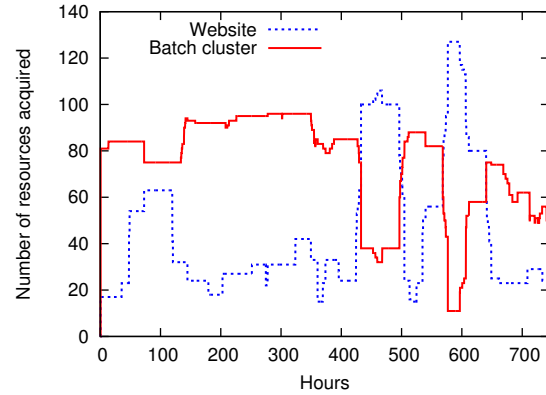
broker implements a simple policy that balances the load evenly among the sites.

We implemented an adaptive service manager that requests resource leases at five-minute intervals to match a changing load signal. We derived sample input loads from traces of two production systems: a job trace from a production compute cluster at Duke, and a trace of CPU load from a major e-commerce website. We scaled the load signals to a common basis. Figure 6 shows scaled cluster resource demand—interpreted as the number of nodes to request—over a one-month segment for both traces (five-minute intervals). We smoothed the e-commerce demand curve with a “flop-flip” filter from [6]. This filter holds a stable estimate of demand $E_t = E_{t-1}$ until that estimate falls outside some tolerance of a moving average ($E_t = \beta E_{t-1} + (1 - \beta)O_t$) of recent observations, then it switches the estimate to the current value of the moving average. The smoothed demand curve shown in Figure 6 uses a 150-minute sliding window moving average, a step threshold of one standard deviation, and a heavily damped average $\beta=7/8$.

Figure 7 demonstrates the effect of varying lease terms on the broker's ability to match the e-commerce load curve. For a lease term of one day, the leased resources closely match the load; however, longer terms diminish the broker's ability to match demand. To quantify the effectiveness and efficiency of allocation over the one-month period, we compute the *root mean squared error* (RMSE) between the load signal and the requested resources. Numbers closer to zero are better: an RMSE of zero indicates that allocation exactly matches demand. For a lease term of 1 day, the RMSE is 22.17 and for a lease term of 7 days, the RMSE is 50.85. Figure 7 reflects a limitation of the pure brokered leasing model as prototyped: a lease holder can return unused resources to the authority, but it cannot return the ticket to the broker to allocate for other purposes.



(a) Lease term of 12 emulated hours.



(b) Lease term of 3 emulated days.

Figure 8: Brokering of 140 machines from two sites between a low-priority computational batch cluster and a high-priority e-commerce website that are competing for machines. Where there is contention for machines, the high priority website receives its demand causing the batch cluster to receive less. Short lease terms (a) are able to closely track resource demands, while long lease terms (b) are unable to match short spikes in demand.

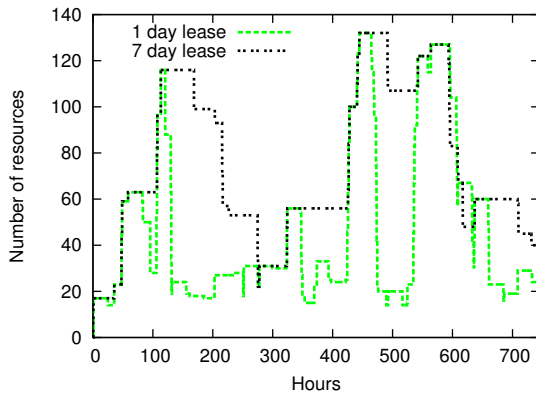


Figure 7: The effect of longer lease terms on a broker's ability to match guest application resource demands. The website's service manager issues requests for machines, but as the lease term increases, the broker is less effective at matching the demand.

To illustrate adaptive provisioning between competing workloads, we introduce a second service manager competing for resources according to the batch load signal. The broker uses FCFS priority scheduling to arbitrate resource requests; the interactive e-commerce service receives a higher priority. Figure 8 shows the assigned slice sizes for lease terms of (a) 12 emulated hours and (b) 3 emulated days. As expected, the batch cluster receives fewer nodes during load surges in the e-commerce service. However, with longer lease terms, load matching becomes less accurate, and some short demand spikes are not served. In some instances, resources assigned to one guest are idle while the other guest saturates but cannot obtain more. This is seen in the RMSE calculated from

N	cluster size
l	number of active leases
n	number of machines per lease
t	term of a lease in virtual clock ticks
α	overhead factor (ms per virtual clock ticks)
t'	term of a lease (ms)
r'	average number of machine reallocations per ms

Table 3: Parameter definitions for Section 5.3

Figure 8: the website has a RMSE of (a) 12.57 and (b) 30.70 and the batch cluster has a RMSE of (a) 23.20 and (b) 22.17. There is a trade-off in choosing the length of lease terms: longer terms are more stable and better able to amortize resource setup/teardown costs improving fidelity (from Section 5.1), but are not as agile to changing demand as shorter leases.

5.3 Scaling of Infrastructure Services

These emulation experiments demonstrate how the lease management and configuration services scale at saturation. Table 3 lists the parameters used in our experiment: for a given cluster size N at a single site, one service manager injects lease requests to a broker for N nodes (without lease extensions) evenly split across l leases (for $N/l = n$ nodes per lease) every lease term t (giving a request injection rate of l/T). Every lease term t the site must reallocate or “flip” all N nodes. We measure the total overhead including lease state maintenance, network communication costs, actor database operations, and event polling costs. Given parameter values we can derive the worst-case minimum lease term, in real time, that the system can support at saturation.

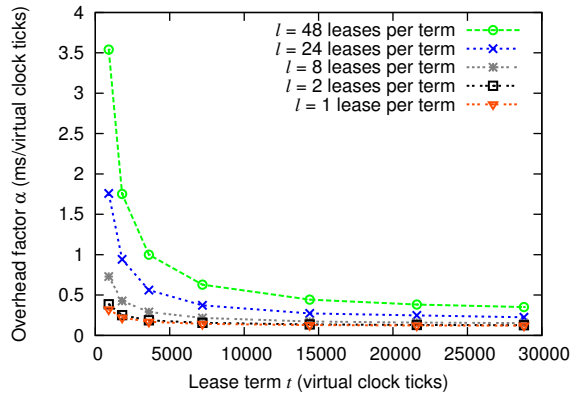


Figure 9: The implementation overhead for an example Shirako scenario for a single emulated cluster of 240 machines. As lease term increases, the overhead factor α decreases as the actors spend more of their time polling lease status rather than more expensive setup/teardown operations. Overhead increases with the number of leases (l) requested per term.

As explained in Section 4.2, each actor's operations are driven by a virtual clock at an arbitrary rate. The prototype polls the status of pending lease operations (i.e., completion of *join/leave* and *setup/teardown* events) on each tick. Thus, the rate at which we advance the virtual clock has a direct impact on performance: a high *tick rate* improves responsiveness to events such as failures and completion of configuration actions, but generates higher overhead due to increased polling of lease and resource status. In this experiment we advance the virtual clock of each actor as fast as the server can process the clock ticks, and determine the amount of real time it takes to complete a pre-defined number of ticks. We measure an *overhead factor* α : the average lease management overhead in milliseconds per clock tick. Lower numbers are better.

Local communication. In this experiment, all actors run on a single x335 server and communicate with local method calls and an in-memory database (no LDAP). Figure 9 graphs α as a function of lease term t in virtual clock ticks; each line presents a different value of l keeping N constant at 240. The graph shows that as t increases, the average overhead per virtual clock tick decreases; this occurs because actors perform the most expensive operation, the reassignment of N nodes, only once per lease term leaving less expensive polling operations for the remainder of the term. Thus, as the number of polling operations increases, they begin to dominate α . Figure 9 also shows that as we increase the number of leases injected per term, α also increases. This demonstrates the increased overhead to manage the leases.

At a clock rate of one tick per second, the overhead represents less than 1% of the latency to prime a node (i.e., to write a new OS image on local disk and boot it). As an example from Figure 9, given this tick rate, for a lease term of 1 hour (3,600 virtual clock ticks), the total over-

N (cluster size)	α	stdev α	t'
120	0.1183	0.001611	425.89
240	0.1743	0.000954	627.58
360	0.2285	0.001639	822.78
480	0.2905	0.001258	1,045.1

Table 4: The effect of increasing the cluster size on α as the number of active leases is held constant at one lease for all N nodes in the cluster. As cluster size increases, the per-tick overhead α increases, driving up the minimal lease term t' .

RPC Type	Database	α	stdev α	t'	r'
Local	Memory	.1743	.0001	627	.3824
Local	LDAP	5.556	.1302	20,003	.0120
SOAP	Memory	27.902	1.008	100,446	.0024
SOAP	LDAP	34.041	.2568	122,547	.0019

Table 5: Impact of overhead from SOAP messaging and LDAP access. SOAP and LDAP costs increase overhead α (ms/virtual clock tick), driving down the maximum node flips per millisecond and driving up the minimum practical lease term t' .

head of our implementation is $t' = t\alpha = 2.016$ seconds with $l=24$ leases per term. The lease term t' represents the minimum term we can support considering only implementation overhead. For COD, these overheads are at least an order of magnitude less than the *setup/teardown* cost of nodes with local storage. From this we conclude that the *setup/teardown* cost, not overhead, is the limiting factor for determining the minimum lease term. However, overhead may have an effect on more fine-grained resource allocation, such as CPU scheduling, where reassignments occur at millisecond time scales.

Table 4 shows the effect of varying the cluster size N on the overhead factor α . For each row of the table, the service manager requests one lease ($l=1$) for N nodes ($N=n$) with a lease term of 3,600 virtual clock ticks (corresponding to a 1 hour lease with a tick rate of 1 second). We report the average and one standard deviation of α across ten runs. As expected, α and t' increase with cluster size, but as before, t' remains an order of magnitude less than the *setup/teardown* costs of a node.

SOAP and LDAP. We repeat the same experiment with the service manager running on a separate x335 server, communicating with the broker and authority using SOAP/XML. The authority and broker write their state to a shared LDAP directory server. Table 5 shows the impact of the higher overhead on t' and r' , for $N=240$. Using α , we calculate the maximum number of node flips per millisecond $r' = N/(T\alpha)$ at saturation. The SOAP and LDAP overheads dominate all other lease management costs: with $N = 240$ nodes, an x335 can process 380 node flips per second, but SOAP and LDAP communication overheads reduce peak flip throughput to 1.9 nodes per second. Even so, neither value presents a limiting factor for today's cluster sizes (thousands of nodes). Using SOAP and LDAP at saturation requires a minimum lease term t' of 122 seconds, which approaches the

setup/teardown latencies (Section 5.1).

From these scaling experiments, we conclude that lease overhead is quite modest, and that costs are dominated by per-tick resource polling, node reassignment, and network communication. In this case, the dominant costs are LDAP access and SOAP operations and the cost for Ant to parse the XML configuration actions and log them.

6 Related Work

Variants of leases are widely used when a client holds a resource on a server. The common purpose of a lease abstraction is to specify a mutually agreed time at which the client's right to hold the resource expires. If the client fails or disconnects, the server can reclaim the resource when the lease expires. The client renews the lease periodically to retain its hold on the resource.

Lifetime management. Leases are useful for distributed garbage collection. The technique of robust distributed reference counting with expiration times appeared in Network Objects [5], and subsequent systems—including Java RMI [29], Jini [27], and Microsoft .NET—have adopted it with the “lease” vocabulary. Most recently, Web Services WSRF [10] has defined a lease protocol as a basis for lifetime management of hosted services.

Mutual exclusion. Leases are also useful as a basis for distributed mutual exclusion, most notably in cache consistency protocols [14, 21]. To modify a block or file, a client first obtains a lease for it in an exclusive mode. The lease confers the right to access the data without risk of a conflict with another client as long as the lease is valid. The key benefit of the lease mechanism itself is availability: the server can reclaim the resource from a failed or disconnected client after the lease expires. If the server fails, it can avoid issuing conflicting leases by waiting for one lease interval before granting new leases after recovery.

Resource management. As in SHARP [13], the use of leases in Shirako combines elements of both lifetime management and mutual exclusion. While providers may choose to overbook their physical resources locally, each offered logical resource unit is held by at most one lease at any given time. If the lease holder fails or disconnects, the resource can be allocated to another guest. This use of leases has three distinguishing characteristics:

- Shirako leases apply to the resources that host the guest, and not to the guest itself; the resource provider does not concern itself with lifetime management of guest services or objects.
- The lease quantifies the resources allocated to the guest; thus leases are a mechanism for service quality assurance and adaptation.
- Each lease represents an explicit promise to the lease holder for the duration of the lease. The notion of a lease as an enforceable contract is important in sys-

tems where the interests of the participants may diverge, as in peer-to-peer systems and economies.

Leases in Shirako are also similar to soft-state advance reservations [8, 30], which have long been a topic of study for real-time network applications. A similar model is proposed for distributed storage in L-bone [3]. Several works have proposed resource reservations with bounded duration for the purpose of controlling service quality in a grid. GARA includes support for advance reservations, brokered co-reservations, and adaptation [11, 12].

Virtual execution environments. New virtual machine technology expands the opportunities for resource sharing that is flexible, reliable, and secure. Several projects have explored how to link virtual machines in virtual networks [9] and/or use networked virtual machines to host network applications, including SoftUDC [18], In Vigo [20], Collective [25], SODA [17], and Virtual Playgrounds [19]. Shared network testbeds (e.g., Emulab/Netbed [28] and PlanetLab [4]) are another use for dynamic sharing of networked resources. Many of these systems can benefit from foundation services for distributed lease management.

PlanetLab was the first system to demonstrate dynamic instantiation of virtual machines in a wide-area testbed deployment with a sizable user base. PlanetLab's current implementation and Shirako differ in their architectural choices. PlanetLab consolidates control in one central authority (PlanetLab Central or PLC), which is trusted by all sites. Contributing sites are expected to relinquish permanent control over their resources to the PLC. PlanetLab emphasizes best-effort open access over admission control; there is no basis to negotiate resources for predictable service quality or isolation. PlanetLab uses leases to manage the lifetime of its guests, rather than for resource control or adaptation.

The PlanetLab architecture permits third-party brokerage services with the endorsement of PLC. PlanetLab brokers manage resources at the granularity of individual nodes; currently, the PlanetLab Node Manager cannot control resources across a site or cluster. PLC may delegate control over a limited share of each node's resources to a local broker server running on the node. PLC controls the instantiation of guest virtual machines, but each local broker is empowered to invoke the local Node Manager interface to bind its resources to guests instantiated on its node. In principle, PLC could delegate sufficient resources to brokers to permit them to support resource control and dynamic adaptation coordinated by a central broker server, as described in this paper.

One goal of our work is to advance the foundations for networked resource sharing systems that can grow and evolve to support a range of resources, management policies, service models, and relationships among resource providers and consumers. Shirako defines one model for how the PlanetLab experience can extend to a wider range

of resource types, federated resource providers, clusters, and more powerful approaches to resource virtualization and isolation.

7 Conclusion

This paper focuses on the design and implementation of general, extensible abstractions for brokered leasing as a basis for a federated, networked utility. The combination of Shirako leasing services and the Cluster-on-Demand cluster manager enables dynamic, programmatic, reconfigurable leasing of cluster resources for distributed applications and services. Shirako decouples dependencies on resources, applications, and resource management policies from the leasing core to accommodate diversity of resource types and resource allocation policies. While a variety of resources and lease contracts are possible, resource managers with performance isolation enable guest applications to obtain predictable performance and to adapt their resource holdings to changing conditions.

References

- [1] Ant, September 2005. <http://ant.apache.org/>.
- [2] P. Barham, B. Dragovic, K. Faser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.
- [3] A. Bassi, M. Beck, T. Moore, and J. S. Plank. The logistical backbone: Scalable infrastructure for global data grids. In *Proceedings of the 7th Asian Computing Science Conference on Advances in Computing Science*, December 2002.
- [4] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *First Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.
- [5] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, December 1993.
- [6] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 103–116, October 2001.
- [7] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *Proceedings of the Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003.
- [8] M. Degermark, T. Kohler, S. Pink, and O. Schelen. Advance reservations for predictive service in the Internet. *Multimedia Systems*, 5(3):177–186, 1997.
- [9] R. J. Figueiredo, P. A. Dinda, and F. Fortes. A case for grid computing on virtual machines. In *International Conference on Distributed Computing Systems (ICDCS)*, May 2003.
- [10] I. Foster, K. Czajkowski, D. F. Ferguson, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. Modeling and managing state in distributed systems: The role of OGSi and WSRF. *Proceedings of the IEEE*, 93(3):604–612, March 2005.
- [11] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, June 1999.
- [12] I. Foster and A. Roy. A quality of service architecture that combines resource reservation and application adaptation. In *Proceedings of the International Workshop on Quality of Service*, June 2000.
- [13] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.
- [14] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.
- [15] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, scalable disk imaging with Frisbee. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.
- [16] D. Irwin, J. Chase, L. Grit, and A. Yumerefendi. Self-Recharging Virtual Currency. In *Proceedings of the Third Workshop on Economics of Peer-to-Peer Systems (P2P-ECON)*, August 2005.
- [17] X. Jiang and D. Xu. Soda: A service-on-demand architecture for application service hosting utility platforms. In *12th IEEE International Symposium on High Performance Distributed Computing*, June 2003.
- [18] M. Kallahalla, M. Uysal, R. Swaminathan, D. Lowell, M. Wray, T. Christian, N. Edwards, C. Dalton, and F. Gittler. SoftUDC: A software-based data center for utility computing. In *Computer*, volume 37, pages 38–46. IEEE, November 2004.
- [19] K. Keahey, K. Doering, and I. Foster. From sandbox to playground: Dynamic virtual environments in the grid. In *5th International Workshop in Grid Computing*, November 2004.
- [20] I. Krsul, A. Ganguly, J. Zhang, J. Fortes, and R. Figueiredo. VM-Plants: Providing and managing virtual machine execution environments for grid computing. In *Supercomputing*, October 2004.
- [21] R. Macklem. Not quite NFS, soft cache consistency for NFS. In *USENIX Association Conference Proceedings*, pages 261–278, January 1994.
- [22] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and Implementation Tradeoffs in Wide-Area Resource Discovery. In *Proceedings of Fourteenth Annual Symposium on High Performance Distributed Computing (HPDC)*, July 2005.
- [23] P. M. Papadopoulos, M. J. Katz, and G. Bruno. NPACI Rocks: Tools and techniques for easily deploying manageable Linux clusters. In *IEEE Cluster 2001*, October 2001.
- [24] J. Pormann, J. Board, D. Rose, and C. Henriquez. Large-scale modeling of cardiac electrophysiology. In *Proceedings of Computers in Cardiology*, September 2002.
- [25] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *5th Symposium on Operating Systems Design and Implementation*, December 2002.
- [26] N. Taesombut and A. Chien. Distributed Virtual Computers (DVC): Simplifying the development of high performance grid applications. In *Workshop on Grids and Advanced Networks*, April 2004.
- [27] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, July 1999.
- [28] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.
- [29] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *Proceedings of the Second USENIX Conference on Object-Oriented Technologies (COOTS)*, June 1997.
- [30] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, 7(5):8–18, September 1993.

Understanding and Validating Database System Administration

Fábio Oliveira, Kiran Nagaraja, Rekha Bachwani,
Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen
Department of Computer Science
Rutgers University, Piscataway, NJ 08854

Abstract

A large number of enterprises need their commodity database systems to remain available at all times. Although administrator mistakes are a significant source of unavailability and cost in these systems, no study to date has sought to quantify the frequency of mistakes in the field, understand the context in which they occur, or develop system support to deal with them explicitly. In this paper, we first characterize the typical administrator tasks, testing environments, and mistakes using results from an extensive survey we have conducted of 51 experienced administrators. Given the results of this survey, we next propose system support to validate administrator actions before they are made visible to users. Our prototype implementation creates a validation environment that is an extension of a replicated database system, where administrator actions can be validated using real workloads. The prototype implements three forms of validation, including a novel form in which the behavior of a database replica can be validated even without an example of correct behavior for comparison. Our results show that the prototype can detect the major classes of administrator mistakes.

1 Introduction

Most enterprises rely on at least one database management system (DBMS) running on commodity computers to maintain their data. A large fraction of these enterprises, such as Internet services and world-wide corporations, need to keep their databases operational at all times. Unfortunately, doing so has been a difficult task.

A key source of unavailability in these systems is database administrator (DBA) mistakes [10, 15, 20]. Database administration is mistake-prone as it involves many complex tasks, such as storage space management, database structure management, and performance tuning. Even worse, as shall be seen, DBA mistakes are

typically not maskable by redundancy (as in an underlying RAID subsystem) or standard fault-tolerance mechanisms (such as a primary-backup scheme). Thus, DBA mistakes are frequently exposed to the surrounding systems, database applications and users, causing unavailability and potentially high revenue losses.

Previous work has categorized DBA mistakes into broad classes and across different DBMSs [10]. However, no previous work has quantified the frequency of the mistakes in the field, characterized the context in which they occur, or determined the relationship between DBA experience and mistakes. Furthermore, no previous work has developed system support to deal with DBA mistakes explicitly.

In this paper, we address these issues in detail. We first characterize (in terms of class and frequency) the typical DBA tasks, testing environments, and mistakes, using results from an extensive survey we have conducted of 51 DBAs with at least 2 years of experience. Our survey responses show that tasks related to recovery, performance tuning, and database restructuring are the most common, accounting for 50% of the tasks performed by DBAs. Regarding the frequency of mistakes, the responses suggest that DBA mistakes are responsible (entirely or in part) for roughly 80% of the database administration problems reported. The most common mistakes are deployment, performance, and structure mistakes, all of which occur once per month on average. These mistakes are caused mainly by the current separation of and differences between testing and online environments.

Given the high frequency of DBA mistakes, we next propose system support to *validate* DBA actions before exposing their effects to the DBMS clients. As we described in [16], the key idea of validation is to check the correctness of human actions in a *validation environment* that is an extension of the online system. In particular, the components under validation, called *masked* components, are subjected to realistic (or even live) workloads. Critically, their state and configurations are not modified

when transitioning from validation to live operation.

In [16], we proposed trace and replica-based validation for Web and application servers. Both techniques rely on samples of correct behavior. Trace-based validation involves periodically collecting traces of live requests and replaying the trace for validation. Replica-based validation involves designating each masked component as a “mirror” of a live component. All requests sent to the live component are then duplicated and also sent to the mirrored, masked component. Results from the masked component are compared against those produced by the live component. Here, we extend our work to deal with DBMSs by modifying a database clustering middleware called Clustered-JDBC (C-JDBC) [7].

Furthermore, we propose a novel form of validation, called *model-based validation*, in which the behavior of a masked component can be validated even when we do not have an example of correct behavior for comparison. In particular, we use model-based validation to verify actions that might change the database structure.

We evaluate our prototype implementation by running a large number of mistake-injection experiments. From these experiments, we find that the prototype is easy to use in practice, and that validation is effective in catching a majority of the mistakes the surveyed DBAs reported. In particular, our validation prototype detected 19 out of 23 injected mistakes, covering all classes of mistakes reported by the surveyed DBAs.

In summary, we make three main contributions:

- We present a wealth of data on the behavior of experienced administrators of real databases. This contribution is important in that actual data on DBA mistakes is not publicly available, due to commercial and privacy considerations.
- We propose model-based validation for the situations when the behavior of the components affected by the DBA actions is supposed to change and there are no instances of correct behavior for comparison.
- We implement a realistic validation environment for dealing with DBA mistakes. We demonstrate the benefits of the prototype through an extensive set of mistake-injection experiments.

The remainder of the paper is organized as follows. The next section describes the related work. Section 3 describes our survey and analyzes the responses we received. Section 4 describes validation and our prototype. Section 5 presents our validation results. In Section 6, we broaden the discussion of the DBA mistakes and the validation approach to a wider range of systems. Finally, Section 7 draws our conclusions.

2 Related Work

Database administration mistakes. Only a few papers have addressed database administrator mistakes in detail. In two early papers [11, 12], Gray estimated the frequency of DBA mistakes based on fault data from deployed Tandem systems. However, whereas today’s systems are mostly built from commodity components, the Tandem systems included substantial custom hardware and software for tolerating single faults. This custom infrastructure could actually mask several types of mistakes that today’s systems may be vulnerable to.

The work of Gil *et al.* [10] included a categorization of administrator tasks and mistakes into classes, and a comparison of their specific details across different DBMSs. Vieira and Madeira [20] proposed a dependability benchmark for database systems based on the injection of administrator mistakes and observation of their impact. In this paper, we extend these contributions by quantifying the frequency of the administrator tasks and mistakes in the field, characterizing the testing environment administrators use, and identifying the main weaknesses of DBMSs and support tools with respect to database administration. Furthermore, our work develops system support to deal with administrator mistakes, which these previous contributions did not address.

Internet service operation mistakes. A few more papers have addressed operator mistakes in Internet services. The work of Oppenheimer *et al.* [17] considered the universe of failures observed by three commercial services. With respect to operators, they broadly categorized their mistakes, described a few example mistakes, and suggested some avenues for dealing with them. Brown and Patterson [4] proposed “undo” as a way to rollback state changes when recovering from operator mistakes. Brown [3] performed experiments in which he exposed human operators to an implementation of undo for an email service hosted by a single node. In [16], we performed experiments with volunteer operators, describing all of the mistakes we observed in detail, and designing and implementing a prototype validation infrastructure that can detect and hide a majority of the mistakes. In this paper, we extend these previous contributions by considering mistakes in database administration and introducing a new validation technique.

Validation. We originally proposed trace and replica-based validation for Web and application servers in Internet services [16]. Trace-based validation is similar in flavor to fault diagnosis approaches [1, 8] that maintain statistical models of “normal” component behavior and dynamically inspect the service execution for deviations from this behavior. These approaches typically focus on the data flow behavior across the systems com-

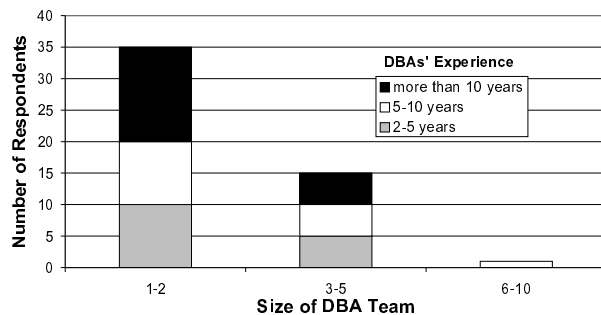


Figure 1: *Distribution of DBAs across team sizes.*

ponents, whereas trace-based validation inspects the actual responses coming from components and can do so at various semantic levels.

Replica-based validation has been used before to tolerate Byzantine failures and malicious attacks, e.g. [5, 6, 13]. In this context, replicas are a permanent part of the distributed system and validation is constantly performed via voting.

Model-based validation is loosely related to two approaches to software debugging: model checking (e.g., [21]) and assertion checking (e.g., [9, 18]). Of these, it is closest to the PSpec system [18] for assertion checking. However, because PSpec was concerned with performance problems as opposed to detecting human mistakes, the authors did not consider structural issues such as component connectivity and database schemas. Besides its focus on human mistakes, model-based validation differs from other assertion-checking efforts (e.g., [9]) in that our assertions are external to the component being validated. Model-based validation differs from model checking in that it validates components dynamically based on their behavior, rather than statically based on their source codes.

In this paper, we extend our work on trace and replica-based validation to database servers, which pose a number of new challenges (Section 6). For example, our previous validation prototype did not have to manage hard state during or after validation. In validating database systems, we need to consider this management and its associated performance and request buffering implications. Furthermore, we propose model-based validation to validate actions without examples of correct behavior.

Other approaches to dealing with mistakes. Validation is orthogonal to Undo [4] in that it hides human actions until they have been validated in a realistic validation environment. A more closely related technique is “offline testing” [2]. Validation takes offline testing a step further by operating on components in an environment that is an extension of the live system. This allows validation to catch a larger number of mistakes as discussed in [16].

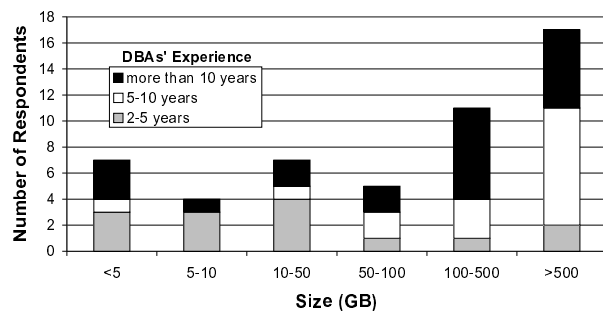


Figure 2: *Distribution of DBAs across database sizes.*

3 Understanding DB Administration

We have conducted an online survey to unveil the most common tasks performed by DBAs, the problems and mistakes that occur during administration, and aspects of the environment in which DBAs carry out their duties. We have posted the survey (available at http://vivo.cs.rutgers.edu/dba_survey.html) to the USENIX SAGE mailing list and the most visible database-related Usenet newsgroups, namely *comp.databases.** and *comp.data.administration*. We next present an analysis of the 51 responses we received.

3.1 Main Characteristics of the Sample

The DBAs who replied to our survey represent a wide spectrum of organizations, DBMSs, experience levels, DBA team sizes, and database sizes. In particular, judging by the DBAs’ email addresses, they all work for different organizations. The DBAs use a variety of DBMSs, including Microsoft SQL Server, Oracle, Informix, DB2, Sybase, MySQL, PostgreSQL, Ingres, IMS, and Progress. The most common DBMSs in our sample are MS SQL Server (31%), Oracle (22%), and MySQL (13%).

The DBAs are highly experienced: 15 of them had between 2 and 5 years of experience, 16 had between 5 and 10 years of experience, and 20 had more than 10 years of experience. Figures 1 and 2 show the sizes of the administration teams to which the DBAs belong, and the sizes of the databases they manage, respectively. The figures show the breakdown of DBAs per experience level.

From Figure 1, it is clear that most of the DBAs work alone or in small groups, regardless of experience level. This result suggests that the size of DBA teams is determined by factors other than DBA experience. Nevertheless, a substantial number of DBAs do work in larger teams, increasing the chance of conflicting actions by different team members. Finally, Figure 2 shows that DBAs of all experience levels manage small and large databases. However, one trend is clear: the least experienced DBAs (2–5 years of experience) tend to manage

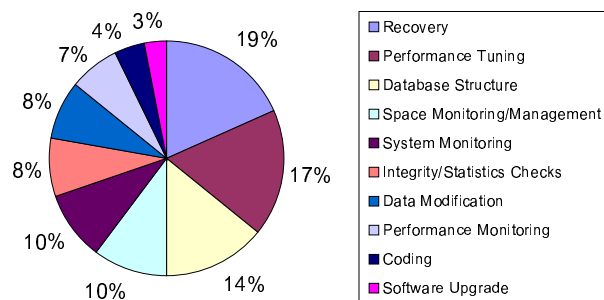


Figure 3: Task categories. The legend lists the categories in decreasing order of frequency.

smaller databases; they represent 12% of the DBAs responsible for databases that are larger than 50 GB, but represent 55% of those DBAs in charge of databases that are smaller than 50 GB.

The experience level of the DBAs who participated in our survey and the diversity in organizations, DBMSs, and team and database sizes suggest that the data we collected is representative of common DBA practices.

3.2 Common DBA Tasks

We asked the DBAs to describe the three most common tasks they perform. Figure 3 shows the categories of tasks they reported, as well as the breakdown of how frequently each category was mentioned out of a total of 126 answers (several DBAs listed fewer than three tasks).

The *Recovery* category corresponds to those tasks that prepare or test the DBMS with respect to recovery operations, such as making backups, testing backups, and performing recovery drills. *Performance Tuning* tasks involve performance optimizations, such as creating or modifying indexes to speed up certain queries, and optimizing the queries themselves. The *Database Structure* tasks involve changing the database schema by adding or removing table columns, or adding or removing entire tables, for example. The *Space Monitoring/Management*, *System Monitoring*, *Performance Monitoring*, and *Integrity/Statistics Checks* tasks all involve monitoring procedures, such as identifying the applications or queries that are performing poorly. The *Data Modification* category represents those tasks that import, export, or modify actual data in the database. Finally, the *Coding* task involves writing code to support applications, whereas the *Software Upgrade* tasks involve the upgrade of the operating system, the DBMS, or the supporting tools.

The three most frequently mentioned categories were Recovery, Performance Tuning, and Database Structure, respectively amounting to 19%, 17%, and 14% of all tasks described by the DBAs. If we collapse the Space Monitoring/Management, System Monitoring, and Per-

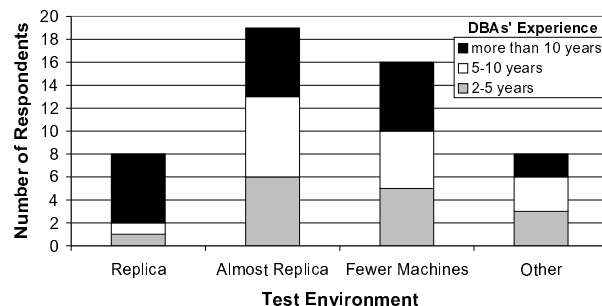


Figure 4: Test environment configurations.

formance Monitoring categories, we can see that 27% of the reported tasks have to do with checking the system behavior. From a different perspective, 24% of all tasks are related to performance, in which case the DBAs are engaged in either identifying the reasons for poor performance (Performance Monitoring) or looking for opportunities to further improve the overall DBMS performance (Performance Tuning). If we consider that checking and updating database statistics are carried out to allow for more accurate optimizations, the fraction of performance-oriented tasks is actually 32%.

Twenty one DBAs (41%) reported that they use third-party support tools for these different tasks. Several of these DBAs actually use multiple types of tools. In more detail, 14 DBAs use tools for Performance Monitoring tasks, 12 DBAs use tools for Recovery tasks, 12 DBAs use tools for Database Structure tasks, 7 DBAs use tools for Performance Tuning tasks, and 7 DBAs use tools for granting/revoking access privileges; the latter tools may be needed in Database Structure, Coding, and Software Upgrade tasks.

3.3 Testing Context

Another important set of questions in our survey concerned the context in which the above tasks are performed. More specifically, we sought to understand the testing methodology that DBAs rely upon to verify the correctness of their actions. The next paragraphs describe our findings with respect to the environment and approach used for testing.

Testing environment. Figure 4 depicts the distribution of test environments, breaking results down with respect to the DBA experience levels. We categorize the environments according to how similar they are to the online database environment: an exact replica (*Replica*), an environment that differs from the online environment only with respect to how powerful the machines are (*Almost Replica*), an environment with fewer machines than the online environment (*Fewer Machines*), or other configu-

Problem category	Average frequency	Number of DBAs per experience level			% of all problems	Caused by DBA mistakes
		2–5 years	5–10 years	> 10 years		
Deployment problems	Once a month	4	3	3	11	most
Performance problems	Once a month	2	2	4	9	most
General structure problems	Once a month	2	4	3	10	most
DBMS problems	Once a month	1	1	2	5	none
Access-privilege problems	Once in 2 months	3	5	1	10	all
Space problems	Once in 2 months	3	3	8	16	some
General maintenance problems	Once in 3 months	6	5	5	18	most
Hardware problems	Once a year	3	6	5	16	none
Data problems	Once a year	–	1	3	5	some

Table 1: *Reported problems, estimated average frequency of occurrence, number of DBAs who mentioned each problem as frequent broken down by DBA experience, percentage of DBAs who mentioned the problem as frequent, and, qualitatively, how often it is caused by DBA mistakes.*

rations (*Other*). Irrespective of the test environment, the test machines are typically loaded with only a fraction of the online database.

We can see that 84% of the DBAs test their actions in environments that are different from the online environment. Further, DBAs of different experience levels use these non-replica testing practices in roughly similar percentages. We conjecture that these practices are a result of the performance and cost implications of using exact replicas for testing. Regardless of the reason, it is possible for actions to appear correct in these environments, but cause problems when migrated to the online environment. Not to mention the fact that the migration itself is mistake-prone, since it is performed by the DBA manually or via deployment scripts.

Another interesting observation is that 8 DBAs use other approaches to testing. Three of these DBAs replicate the online databases on the online machines themselves, and use the replicated databases as test instances. This approach is problematic when the DBA actions involve components shared by online and testing environments, e.g. the operating system or the I/O devices.

Two of the most experienced DBAs also mentioned a well-structured testing environment comprising three sets of machines: (1) “development machines” used for application developers to ascertain that a particular application interacts with the necessary databases as expected; (2) “integration machines”, which host all applications and are subjected to more aggressive tests, including the use of load generators; (3) “quality-assurance machines” used for ensuring that the system conforms with established standards before it can be deemed deployable.

Testing approach. Another important issue is how DBAs test modifications to existing databases and newly designed databases before deployment to the production system. Most DBAs (61%) report that they perform testing manually or via their own scripts. Two other DBAs

report that, depending upon the nature of the actions to be performed, they do not carry out any offline testing whatsoever. Finally, 2 DBAs reported testing by means of documented operability standards that specify a set of requirements to be satisfied before deployment to production. In such cases, to determine if the requirements are obeyed, performance tests, design reviews, and security analyses are carried out.

3.4 Problems in DB Administration

We asked the DBAs to describe the three most frequent problems that arise during database administration. Based on the descriptions they provided, we derived the categories listed in Table 1. The table shows how frequently (on average) these DBAs estimate each problem category to occur, how many DBAs with different experience levels alluded to the category, the percentage of DBAs who mentioned the category, and qualitatively how often a problem in the category is caused by a DBA mistake. In the following paragraphs, we describe the problems, their causes, and how they affect the system.

Deployment problems. This category of problems occurs when changes to the online system cause the database to misbehave, even though the changes may have been tested in an offline testing environment. These problems occur in DBA tasks that involve migrating changes from a testing environment to the online environment (i.e., Performance Tuning, Database Structure, Data Modification, Coding, and Software Upgrade tasks), and are typically due to DBA mistakes committed during this migration process. Specifically, the DBAs reported a number of causes for these problems: (1) bugs in the DBA’s deployment scripts, which are aggravated by the DBMSs’ typically poor support for debugging and logging changes; (2) DBAs forget to change the structure of the online database before deploying a new or recently

modified application; (3) DBAs accidentally propagate the changes made to the database in the testing environment to the online system; (4) DBAs make inappropriate changes directly to the online database; (5) DBAs forget to reapply indexes in the production database; and (6) applications compiled against the wrong database schema are deployed online.

Note that causes (1)–(5) are DBA mistakes. Some of them affect the interaction between the database and the applications. If the deployed database and the applications are not consistent, non-existent structures might be accessed, thus generating fatal SQL errors.

As shown in Table 1, deployment problems occur frequently (once a month on average), according to the 10 DBAs who mentioned this category. Out of all DBAs, 40% of them mentioned the lack of integrated versioning control for the database and its applications as the main weakness of current DBMSs and third-party tools with respect to deployment. Another 27%, 13%, and 13% mentioned the complexity of the DBMSs and tools, poor support for comparisons between test and online environments, and the burden posed by interdependencies between database objects, respectively.

Performance problems. Typically, when the DBMS delivers poor performance to applications or users, the culprit is the DBA, the application developer, or both. Two DBA mistakes compromising the DBMS performance were mentioned: (1) erroneous performance tuning, in the face of the plethora of configuration parameters offered by DBMSs; and (2) inappropriate database design, including database object structures and indexing scheme. These mistakes can occur in DBA tasks that involve performance tuning, removing/adding database objects, or changing the software (i.e., Performance Tuning, Database Structure, Coding, and Software Upgrade tasks). On the application developer side, the DBAs complained about poorly designed queries that take long to complete and consume a lot of resources.

As shown in Table 1, performance problems happen frequently (once a month on average), according to 8 DBAs. In fact, 4 of these DBAs have more than 10 years of experience and consistently commented on DBA-induced poor performance in particular.

General structure problems. Pertaining to this category are incorrect database design and unsuitable changes to the database, both produced by the DBA during Database Structure tasks and leading to malformed database objects, and ill-conceived code on the application developer's part. The DBAs mentioned two particular instances of incorrect database design in our survey: duplicated identity columns and columns too small to hold a particular type of data. Four DBAs also mentioned that mistakes in database design and poor appli-

cation code are responsible for deadlocks they have observed. In fact, these 4 DBAs observe deadlocks very frequently, once in two weeks on average.

As Table 1 shows, general structure problems happen frequently (once a month on average), according to the 9 DBAs who mentioned them.

DBMS problems. Four DBAs were victims of bugs in DBMSs. Three of them said that the bugs had only minor impacts on the database operation, but the other said that a DBMS bug was the cause of an outage that lasted half a day. DBMS bugs were not mentioned by many DBAs, but the 4 DBAs who did mention them claim that these bugs occur once a month on average.

Access-privilege problems. Another category of problems affects the privileges to access the database objects. These problems can occur in DBA tasks that involve removing/adding database objects or changing the software (i.e., Database Structure, Coding, and Software Upgrade tasks). According to the DBAs, these problems are caused by two types of mistakes: (1) DBAs do not grant sufficient rights to users or applications, resulting in their inability to access the whole (or parts of the) database; and (2) DBAs grant excessive privileges to some users or applications. Obviously, the latter situation causes a serious security vulnerability.

According to the 9 DBAs who mentioned this category, access-privilege problems also occur frequently (once in two months on average). Interestingly, 1 DBA who mentioned this category uses a third-party tool specifically for granting/revoking access privileges to database objects.

Space problems. This category consists of disk space exhaustion and tablespace (i.e., the space reserved for a set of tables and indexes) problems. These problems are most serious when the DBA fails to monitor and manage the space appropriately (i.e., Recovery and Space Monitoring/Management tasks). Disk space exhaustion is caused chiefly by growing transaction logs, alert logs, and the like. Some DBAs mentioned that the unpredictability of the application users' behavior makes it difficult to foresee a disk space shortage.

Some DBAs also reported tablespaces unexpectedly filling up. Further, a few DBAs mentioned the impossibility of extending a completely used tablespace. Another tablespace-related problem occurs in the context of tablespace defragmentation, an operation that DBAs perform to prevent performance degradation on table accesses. A typical procedure for defragmenting a tablespace involves exporting the affected tables, dropping them, and re-importing them. During the defragmentation, 1 DBA was unable to re-import the tables due to a bug in the script that automated the procedure and, as a result, the database had to be completely restored.

According to the 14 DBAs who mentioned space problems, they occur once in two months on average.

General maintenance problems. This is the category of problems that DBAs mentioned most frequently; 16 DBAs mentioned it. These problems occur during common maintenance tasks, such as software or hardware upgrades, configuring system components, and managing backups. Regarding upgrades, some DBAs alluded to failures resulting from incompatibilities that arose after such upgrades. Other DBAs mentioned mistakes such as the DBA incorrectly shutting down the database and the DBA forgetting to restart the DBMS replication capability after a shutdown. In terms of configuration, 3 DBAs described situations in which the DBMS was unable to start after the DBA misconfigured it trying to improve performance. Regarding the management of backups, DBAs listed faulty devices and insufficient space due to poor management.

As Table 1 shows, general maintenance problems happen frequently (once in three months on average), according to the DBAs.

Hardware and data problems. Hardware failure and data loss are the least frequent problems according to the DBAs; they occur only once a year on average. 14 DBAs mentioned hardware failures, whereas only 4 mentioned data loss as a problem.

3.5 Summary and Discussion

We can make several important observations from the data described above:

1. Recovery, performance, and structure tasks are the most common tasks performed by DBAs. Several categories of tasks, including performance tuning, database restructuring, and data modification typically require the DBA to perform and test actions offline and then migrating or deploying changes to the online system.

2. The vast majority of the DBAs test their actions manually (or via their own scripts) in environments that are not exact replicas of the online system. The differences are not only in the numbers and types of machines, but also in the data itself and the test workload.

3. A large number of problems occur in database administration. The most commonly cited were general maintenance, space, and hardware problems. However, deployment, performance, and structure problems were estimated by several DBAs to be the most frequent.

4. Qualitatively, DBA mistakes are the root cause of *all or most* of the deployment, access-privilege, performance, maintenance, and structure problems; these categories represent 58% of the reported problems. They are also responsible for *some* of the space and data problems, which represent 21% of the reported problems. Unfortunately,

the DBA mistakes are typically not maskable by traditional high-availability techniques, such as hardware redundancy or primary-backup schemes. In fact, the mistakes affect the database operation in a number of ways that may produce unavailability (or even incorrect behavior), including: (1) data becoming completely or partially inaccessible; (2) security vulnerabilities being introduced; (3) performance being severely degraded; (4) inappropriate changes and/or unsuitable design giving scope for data inconsistencies; and (5) careless monitoring producing latent errors.

5. DBAs of all experience levels make mistakes of all categories, even when they use third-party tools.

6. The differences and separation between offline testing and online environments are two of the main causes of the most frequent mistakes. Differences between the two environments can cause actions to be correct in the testing system but problematic in the online system. Applying changes that have already been tested in a testing environment to the online system is often an involved, mistake-prone process; even if this process is scripted, mistakes in writing or running the scripts can harm the online system.

These observations lead us to conclude that DBA mistakes have to be addressed for consistent performance and availability. We also conclude that DBAs need additional support beyond what is provided by today's offline testing environments. Thus, we next propose validation as part of the needed infrastructure support to help DBAs reduce the impact of mistakes on system performance and availability. As we demonstrate later, validation hides a large fraction of these mistakes from applications and users, giving the DBA the opportunity to fix his/her mistakes before they become noticeable, as well as eliminates deployment mistakes.

4 Validation

In this section, we first briefly review the overall validation approach. This review is couched in the context of a three-tier Internet service, with the third tier being a DBMS, to provide a concrete example application surrounding the DBMS. Then, we discuss issues that are specific to validating DBMSs and describe our prototype implementation. We close the section with a discussion of the generality and limitations of our approach.

4.1 Background

A validation environment should be as closely tied to the online environment as possible to: (1) avoid latent errors that escape detection during validation but become activated in the online system because of differences between the validation and online environments;

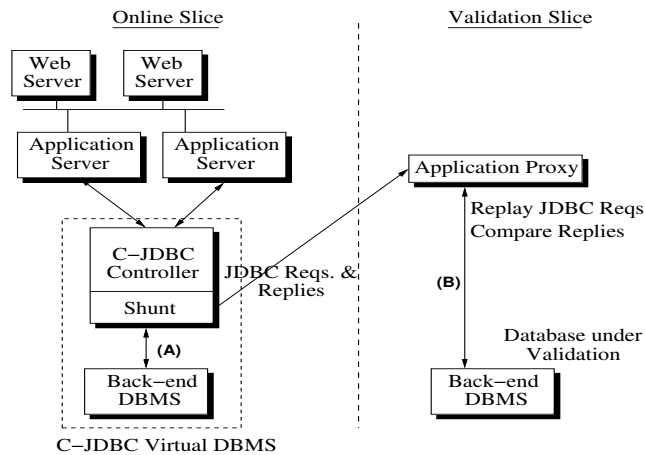


Figure 5: Validation for a three-tier Internet service. In the figure, the two back-end databases are mirrored replicas. The database node in the validation slice is undergoing validation after the DBA has operated on it.

(2) load components under validation with as realistic a workload as possible; and (3) enable operators to bring validated components online without having to change any of the components' configurations, thereby minimizing the chance of new operator mistakes. On the other hand, the components under validation, which we shall call *masked* components for simplicity, must be *isolated* from the online system so that incorrect behaviors cannot cause system failures.

To meet the above goals, we actually host the validation environment on the online system itself. In particular, we divide the components into two logical slices: an online slice that hosts the online components and a validation slice where components can be validated before being integrated into the online slice. Figure 5 shows this validation architecture when a component of the DBMS tier is under validation. To protect the integrity of the online service without completely separating the two slices (which would reduce the validation slice to an offline testing system), we erect an isolation barrier between the slices but introduce a set of connecting *shunts*. The shunts duplicate requests and replies (i.e., inputs and outputs) passing through the interfaces of the components in the live service. Shunts either log these requests and replies or forward them to the validation slice.

We then build a validation harness consisting of *proxy components* that can be used to form a virtual service around the masked components; Figure 5 shows an application proxy being used to drive a masked DBMS. Together, the virtual service and the duplication of requests and replies via the shunts allow operators to validate masked components under realistic workloads. In particular, the virtual service either replays previously recorded logs or accepts forwarded duplicates of live re-

quests and responses from the shunts, feeds appropriate requests to the masked components, and verifies that the outputs of the masked components meet certain validation criteria. Proxies can be implemented by modifying open source components or wrapping code around proprietary software with well-defined interfaces.

Finally, the harness uses a set of *comparator functions*, which compute whether some set of observations of the validation service match a set of criteria. For example, in Figure 5, a comparator function might determine if the streams of requests and replies going across the pair of connections labeled (A) and (B) are similar enough to declare the masked database as working correctly. If any comparison fails, an error is signaled and the validation fails. If after a threshold period of time all comparisons match, the component is considered validated.

Given the above infrastructure, validation becomes conceptually simple. First, a script places the set of components to be worked on in the validation environment, effectively masking them from the live service. The operator then acts on the masked components just as he/she would in the live service. Next, another script instructs the validation harness to surround the masked components with a virtual service, load the components, and check their correctness. If the masked components pass this validation, the script calls a migration function that fully integrates the component into the live service.

4.2 Validation Strategies

In [16], we proposed two validation approaches: *trace-based* and *replica-based* validation. In trace-based validation, for each masked component to be validated, requests and replies passing through the shunts of an equivalent live component are logged and later replayed. During the replay, the logged replies can be compared to the replies produced by the masked component. In replica-based validation, the current offered load on the live service is used, where requests passing through the shunts of an equivalent live component are duplicated and forwarded in real-time to the validation harness to drive the masked component. The shunts also capture the replies generated by the live component and forward them to the harness, which compares them against the replies coming from the masked component.

Unfortunately, trace-based and replica-based validation are only applicable when the output of a masked component can be compared against that of a known correct instance. Many operator actions can correctly lead to a masked component behaving differently than all current/known instances, posing a bootstrapping problem. An example in the context of databases is a change to the database schema (a task that is cited as one of the most common DBA tasks in our survey). After the DBA

changes the schema (e.g., by deleting a column) in the validation environment, the masked database no longer mirrors the online database and so may correctly produce different answers to the same query. The same applies to a previously collected trace.

We propose **model-based validation** to deal with this bootstrapping problem. The key idea behind model-based validation is that a service, particularly one with components that have just been acted on by an operator, should conform to an explicit model. Thus, in model-based validation, we require an explicit representation of the intended consequences of a set of operator actions in a model of the system, and then validate that the dynamic behavior of the masked component matches that predicted by the model.

For example, a simple model for a service composed of a front-end load balancer and a set of back-end servers could specify an assertion to the effect that the resource utilization at the different back-end nodes should always be within a small percentage of each other. This simple model would allow us to validate changes to the front-end device even in the context of heterogeneous servers.

If we can conveniently express these models and check them during validation, we can validate several classes of operator actions that cannot be tackled by trace or replica-based validation. We envision a simple language that can express models for multiple components, including load balancers, firewalls, and servers.

4.3 Implementing Validation for DBMSs

We now describe our prototype validation environment for a service with replicated databases. A replicated database framework allows DBAs to operate on the masked DBMS and validate it while the online DBMS is still servicing live requests, an important property for systems that must provide 24x7 availability.

Our implementation leverages the C-JDBC database clustering middleware [7]. Briefly, C-JDBC clusters a collection of possibly heterogeneous DBMSs into a single virtual DBMS that exposes a single database view with improved scalability and dependability. C-JDBC implements a software controller between a JDBC application and the back-end DBMSs. The controller comprises a request scheduler, a load balancer, and a recovery log. C-JDBC supports a few data distribution schemes. Our prototype uses only one: full data replication across the DBMSs comprising a virtual DBMS. Under full replication, each read request is sent to one replica while writes are broadcast to all replicas.

As shall be seen, critical to our implementation is C-JDBC's capability to disable and disconnect a back-end DBMS, ensuring that its content is a consistent checkpoint with respect to the recovery log, and later reintegrate

this DBMS by replaying the log to update its content to the current content of the virtual database.

As shown in Figure 5, our prototype relies on the C-JDBC controller only in the online slice; the application proxy contacts the database under validation directly. We implement the isolation barrier between the online and validation slices at the granularity of an entire node by running nodes over a virtual network created using Mendosus [14], a fault-injection and network-emulation tool for clustered systems. The general idea is to use Mendosus to partition the virtual network into two parts so that online nodes can see each other but not those in the validation slice and vice-versa. Critically, however, Mendosus can migrate a node between the two parts of the virtual network without requiring any change to the node's networking parameters.

The shunting of requests and responses takes place inside the C-JDBC controller, as can be seen in Figure 5. A critical aspect to be observed when implementing a database shunt is that of ordering. Suppose that the controller has three requests to dispatch within a single transaction: two read requests, *R1* and *R2*, and one write request, *W*. If the order in which the online database executes the requests is *R1*, *W*, and *R2*, whereas the database under validation executes *R1*, *R2*, and then *W*, the database replicas will report different responses for *R2* if the write request modifies the data read by *R2*. If a following request depends on *R2*, the situation becomes even worse. This undesirable ordering mismatch can not only trigger false positives during validation, but also, and more seriously, corrupt the database state.

A logical conclusion from this scenario is the paramount need to enforce a partial order of request execution that both the online database replicas and the database under validation must abide by. A number of consecutive read requests can be executed in any order, but blocks of reads separated by writes (or commits) must be executed in the same order with respect to the write (or commit) requests.

Although the C-JDBC controller guarantees that the mirrored databases connected to it are kept consistent with respect to each transaction independently, it does not guarantee ordering across transactions. In other words, in the face of concurrent transactions, the controller cannot determine whether the back-end databases will execute a read request from one transaction before or after a write request from another transaction. The identification numbers that the controller assigns to requests have nothing to do with actual order of execution. For this reason, we had to modify the controller code by introducing a multiple-reader-single-writer lock used only while requests are being shunted. This lock is used to ensure that each write is executed by itself on the back-end DBMSs, providing a complete ordering between all pairs

of read-write and write-write operations. By using this extra lock and capturing requests right before the controller issues them to the back-ends, our shunting code forwards/collects requests in an order that the application proxy can rely on. During validation, requests can be replayed in this order irrespective of the transactions to which they belong.

We have implemented all three validation strategies and two comparator functions, a reply exact-match and a latency-match comparator. We describe the important details of these implementations next.

Trace-based validation. Trace-based validation requires collecting traces from the online system. Given the above infrastructure, collecting a trace is simple: we ask the C-JDBC controller to create a copy of the database on a “trace disk” and to start logging requests and replies to it serialized to after the copy.

At a later point, suppose that the DBA wants to operate on one of two back-end DBMSs within a C-JDBC virtual DBMS, say to create an index to improve performance, and then use trace-based validation to validate his/her actions. He/she would run two scripts, one before operating on each DBMS and the other after doing so:

Script 1 – take the DBMS to be worked on offline. (1) Instruct the C-JDBC controller to checkpoint and disable the appropriate DBMS node. This means that the disabled database is consistent with a particular point in the controller’s replay log. (2) Instruct Mendosus to move the masked DBMS into the validation slice. (3) Save the masked DBMS’s current state to a persistent backup.

Script 2 – validate the modified DBMS and move it to the online slice. (1) Initialize the DBMS with the database on the trace disk. (2) Start the application proxy. (3) Once validation completes successfully, reinitialize the DBMS with the backup saved in script 1. (4) Instruct Mendosus to move the masked DBMS back to the online slice. (5) Attach the newly modified and validated DBMS to the C-JDBC controller and instruct the controller to integrate it. If validation fails in step (3), the DBA needs to fix any mistakes, and re-start the script.

Replica-based validation. Replica-based validation is quite similar to trace-based validation and can also be run using two scripts. Script 1 is the same as that used for trace-based validation. Script 2 performs the actions enumerated next. (1) Instruct Mendosus to move the masked DBMS back into the online slice, and instruct the controller to bring the masked DBMS up-to-date using its replay log, while keeping the masked DBMS in a *disabled state*. (We actually had to modify the controller to implement this functionality, since the controller would automatically enable a DBMS node after replaying the log.) (2) Now that the masked DBMS is again an exact replica of the online DBMS, start buffering writes

(and commits) to the online virtual DBMS, migrate the masked DBMS back to the validation slice, and start the application proxy. (3) Enable the shunting of requests and replies to the application proxy, and restart write (and commit) processing on the C-JDBC controller. (4) Once validation completes successfully, instruct the controller to halt and buffer all incoming requests; migrate the masked DBMS to the online slice and let it connect to the C-JDBC controller. Note that the two DBMSs have exactly the same content at this point, which makes this reintegration quite fast. (5) Finally, instruct the C-JDBC controller to start processing requests again. If validation fails in step 4, the masked DBMS will remain in the validation slice and will be initialized with the state saved in script 1, so that it can be validated again after the DBA fixes any mistakes.

Note that script 2 forces the controller to buffer write (and commit) requests to the virtual DBMS for a very short period in replica-based validation; just enough time to migrate the masked DBMS to the validation slice and start the application proxy. Because these operations can be performed in only a few milliseconds, the amount of buffering that takes place is typically very small.

Model-based validation. Our vision is to allow the DBA to specify his/her actions in a simple canonical form and associate a small set of assertions with each action. The system can then validate the actions that are actually performed on the masked DBMS by checking that the assertions hold. The specification of actions in the canonical form should be *much simpler* than the actual execution of these actions (say, as SQL queries and commands). They should also be independent of specific implementations of DBMS, which is important because each implementation uses a different variant of SQL. This simplicity and portability are the main advantages of model-based validation. For example, automatically executing the needed actions from the canonical descriptions would require extensive implementations for all possible DBMSs C-JDBC can use.

We have prototyped a simple model-based validation strategy as a proof-of-concept. Our implementation currently focuses on database structure changes, since these tasks were identified as very frequent by the DBAs we surveyed. It includes four canonical actions: add a table, remove a table, add a column, and remove a column. It also defines a set of assertions that must be true about the database schema after an action has been performed compared to the schema before the action. For example, one of the assertions states that, if the DBA will add a table, the schema after the action should contain all the tables in the previous schema plus the table just created.

While this prototype implementation is quite simple, it is also powerful. The reason is that each canonical

action, such as adding a column, can correspond to a lengthy set of real actions. For example, adding a column to the middle of a table might be quite complicated depending on the DBMS being used [15]. This operation might involve, among many other things, unloading the data from the table and dropping it (which would in turn drop all indexes and views associated with the table); recreating the table with the new column; repopulating the table; recreating all necessary indexes and views; and checking if the application programs work correctly with the modified table. In model-based validation, we are only concerned with the model of the database schema. Thus, the operator might specify his/her action as “I will add a column to Table T between two existing columns, A and B.” This allows model-based validation to check that, indeed, in the new schema, table T has one more column that is between A and B.

To address another common set of mistakes described by the DBAs, namely mismatches between applications and the database structure, we combine model-based validation of the DBMSs with trace-based validation of the applications to check that the applications have been updated to correctly deal with the new schema.

In detail, the whole validation process for actions that change the database structure proceeds as follows. First, a script moves the DBMS to the validation slice, asks the DBA to describe his/her intended actions in canonical form, extracts the current schema from the DBMS, and then allows the DBA to act on the masked DBMS. Once the DBA completes the necessary actions, a second script uses model-based validation to check that the corresponding assertions hold. Finally, the DBA brings any application that depends on the database into the validation slice and validates that it works correctly with the new schema using a trace. In the context of the 3-tier Internet service shown in Figure 5, this means that the DBA would move each of the application nodes into the validation slice for validation.

There are three subtle issues that must be addressed when the database schema changes. To make the description of the issues concrete, suppose that we have a service as in Figure 5. First, if a database schema change requires changes to the application servers, then once updated, these servers cannot be returned to the online slice until the new DBMS has been deployed. Second, it may not be possible to properly validate replies from the application servers against replies that were previously logged using a strict comparator, such as exact content matching, when the application servers need to change. Replica-based validation can be used but only after the changes to at least one application server have been validated. Third, during DBMS reintegration into the online system, it may not be possible to replay writes that have been executed on the online system while the masked

DBMS was being changed and validated (e.g., writes that depend on the data in a column that has been removed).

To deal with the first issue, the validation of the application servers and DBMSs needs to occur in two phases as follows. During a period of low load, the DBA can move one DBMS into the validation slice, change the schema, use model-based validation to check the correctness of his/her changes, then move 1/2 of the application servers over to the validation slice, update them as necessary, and use trace-based validation to check that they work correctly with the modified DBMS. After validation is completed, he/she can temporarily halt and buffer requests from the first tier, move all validated components back online, and move the remaining unmodified application servers and DBMS into the validation slice. In essence, this is the point where the live service is changed from operating on the old database schema to the new schema.

To deal with the second issue, we observe that it is possible to validate the application servers using traces collected previously in situations where we know that the changes in schema should not cause SQL fatal errors in the application servers. In this case, a previous trace can be used together with a comparator function that disregards the content of the application server replies but checks for fatal SQL exceptions. (In fact, replica-based validation could also be applied using this weaker comparator function.) In case an exception is found, the validation process fails. When schema changes should cause these exceptions in application servers, a synthetic application server trace needs to be generated that should not cause exceptions with the new schema. Again, an exception found during the validation process would mean a potential DBA mistake.

Finally, to deal with the third issue, our implementation denies writes to the online virtual DBMS when the database structure needs to be changed. This behavior is acceptable for the system we study (an online auction service). To avoid denying writes, an alternative would be to optimistically assume that writes that cannot be replayed because of a change in schema will not occur. In this approach, the C-JDBC controller could be modified to flag an error during reintegration, if a write cannot be replayed. If such an error occurred, it would be up to the DBA to determine the proper course of action.

4.4 Summary

In summary, we find that our three validation strategies are complementary. Trace-based validation can be used for checking the correctness of actions for corner cases that do not occur frequently. Replica-based validation can be used to place the most realistic workload possible on a masked component — the current workload of

the live system. Trace-based and replica-based validation allow the checking of performance tuning actions, such as the creation or modification of indexes. Finally, structural changes can be validated using a combination of model-based and trace-based validation.

With respect to the mistakes reported by DBAs, validation in an extension of the online system allows us to eliminate deployment mistakes, whereas trace, replica, and model-based validation deal with performance tuning and structural mistakes. These three categories of mistakes are the most frequent according to our survey.

5 Evaluation

In this section, we first evaluate our validation approach using a set of mistake-injection benchmarks. We then assess the performance impact of our validation infrastructure on a live service using a micro-benchmark.

5.1 Experimental Setup

Our evaluation is performed in the context of an online auction service modeled after EBay. The service is organized into 3 tiers of servers: Web, application, and database tiers. We use one Web server machine running Apache and three application servers running Tomcat. The database tier comprises one machine running the C-JDBC controller and two machines running back-end MySQL servers (that are replicas of each other within a single C-JDBC virtual DBMS). All nodes are equipped with a 1.2 GHz Intel Celeron processor and 512 MB of RAM, running Linux with kernel 2.4.18-14. The nodes are interconnected by a Fast Ethernet switch.

A client emulator is used to exercise the service. The workload consists of a “bidding mix” of requests (94% of the database requests are reads) issued by a number of concurrent clients that repeatedly open sessions with the service. Each client issues a request, receives and parses the reply, “thinks” for a while, and follows a link contained in the reply. A user-defined Markov model determines which link to follow. During our mistake-injection experiments, the overall load imposed on the system is 60 requests/second, which is approximately 70% of the maximum achievable throughput. The code for the service, the workload, and the client emulator are from the DynaServer project [19].

5.2 Mistake-injection Experiments

We injected DBA mistakes into the auction service. Specifically, we have developed a number of scripts, each of which emulates a DBA performing an administration task on the database tier and contains one mistake that may occur during the task. The scripts are motivated by

Problem Category	# Mistakes Injected	# Mistakes Caught
Deployment	4	4
Performance	2	1
General structure	6	5
DBMS	0	–
Access-privilege	2	1
Space	1	1
General maintenance	6	5
Hardware	0	–
Data	2	2

Table 3: Coverage in mistake-injection experiments.

our survey results and span the most commonly reported tasks and mistake types. The detailed actions and mistakes within the tasks were derived from several database administration manuals and books, e.g. [15]. Table 2 lists the mistakes we injected in our experiments categorized by DBA task and problem (see Section 3). Note that we only designed mistakes for problem categories where at least some problems were reported as originating from DBA mistakes (those marked with some, most, or all for “Caused by DBA mistakes” in Table 1).

Table 3 lists the total number of scripts (mistakes) for each problem category reported in our survey and the number of mistakes caught by validation. Overall, validation detected 19 out of 23 injected mistakes.

It is worth mentioning how validation caught the performance mistake “insufficient number of indexes” and the deployment mistake “indexes not reapplied”. In both cases, a performance degradation was detected by the performance comparator function. The comparator function uses two configurable thresholds to decide on the result of validation: the maximum acceptable execution time difference for each request (set to 60 seconds in our experiments), and the maximum tolerable execution time difference accumulated during the whole validation (set to 30 seconds times the number of requests used in the validation). In our experiments, which were configured to execute 10,000 requests for validation, the absence of an index caused the execution time difference for 13 requests to be greater than 70 seconds. The average time difference among these 13 requests was 32 minutes, and the total time difference over all 10,000 requests was 7.5 hours. Note that validation could have stopped long before the 10,000 requests were executed. However, to see the complete impact of the mistakes, we turned off a timeout parameter that controls the maximum time that each request is allowed to consume during validation.

Our implementation of model-based validation caught 4 out of 5 mistakes from the General structure category for which model-based validation is applicable. We did not catch the “insufficient column size assumed by

Task Category	Problem Category	Details of Mistakes Injected Using Scripts
DB Structure	Deployment	Mismatch in schema used by application and actual database schema
		Unintended modifications to the database schema
		Indexes not reapplied after modifications to structure
	General structure	Insufficient column size assumed by database schema
		Wrong table dropped
		New column given an incorrect name
		Wrong column removed
		Name of existing table incorrectly changed
		Deadlocks caused by erroneous application programming
	Access-privilege	Access to certain tables not granted
		Excess privileges granted
Space Management	Space	Misconfigured autoextension parameter: maximum data file size too small
	Deployment	Incomplete data reimport during a defragment operation
	General maintenance	Data file parameters incorrectly configured: size and path (2 mistakes)
		Log and data files mistakenly deleted (2 mistakes)
Software Upgrade	General maintenance	Incorrect data reloaded post-upgrade, e.g., due to mistakes in transforming data during migration to a different database (from Oracle to MySQL)
Performance Tuning	Performance	Misconfiguration of buffer pool: size is too small
		Insufficient number of indexes
Data Modification	Data	Unchecked data loss/corruption resulting in loading of incomplete/incorrect data into production system (2 mistakes)
Recovery	General maintenance	Incomplete backup due to erroneous backup scripts or inattentive space management, resulting in incorrect/incomplete data during recovery

Table 2: *Operator mistake fault load used in evaluating validation.*

database schema” mistake because it currently does not include the notion of size. However, this can easily be added to a more complete implementation. The 6th structural mistake, “deadlocks caused by erroneous application programming,” could not have been caught by model-based validation because the mistake occurs at the application servers. This mistake was caught by multi-component trace-based validation, however.

We considered the performance mistake “buffer pool size too small” not caught. In our experiment, we changed the MySQL buffer pool size from 256 MB to 40 MB. Regarding its performance impact, the execution time difference for 7 out of 10,000 requests was greater than 1.5 seconds. Had the threshold been set to at most 1.5 seconds, validation would have caught this mistake. This highlights how important it is to specify reasonable thresholds for the performance comparator function.

The second mistake that validation could not catch was “excess privileges granted”, a latent mistake that makes the system vulnerable to unauthorized data access. Validation is not able to deal with this kind of situation because the live (or logged) requests used to exercise the masked components cannot be identified as illegitimate once their authentication has succeeded.

The last mistake that validation overlooked was “log files mistakenly deleted”. The reason is that the DBMS did not behave abnormally in the face of this mistake.

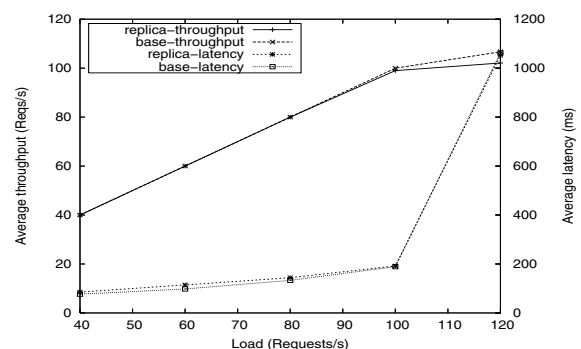


Figure 6: *Performance impact of validation.*

5.3 Performance Overheads

Having shown that validation is effective at masking database administration mistakes, we now consider the performance impact of validation on our auction service.

Shunting. We start by considering the overhead of shunting C-JDBC requests and replies. To expose this overhead, we ran the two back-end database servers on more powerful machines (2.8-GHz Xeon-based machines, each with at least 1 GByte of memory and a 15K-rpm disk) in these experiments.

Figure 6 depicts the average service throughputs (left axis) and average request latencies (right axis) for a system performing replica-based validation and a base sys-

tem that does not shunt any requests or replies, as a function of the offered load. The throughputs and latencies are measured at the client emulator. Note that we do not present results for trace-based validation, since the overhead of logging requests and replies is smaller than that of forwarding them across the isolation barrier.

These results show that the overhead of replica-based validation is negligible in terms of request latency, across the entire range of offered loads. With respect to throughput, the overhead is also negligible until the C-JDBC controller approaches saturation at 120 requests/second; even at that point, the throughput loss is only around 5%. The loss is due to the additional CPU utilization caused by forwarding. In more detail, we find that forwarding imposes an additional 6–10% to the CPU utilization at the controller, across the range of offered loads (logging imposes 3–5% only). In contrast, shunting imposes a load of less than 1 MB/second on network and disk bandwidth, which is negligible for Gigabit networks and storage systems.

Database state handling. We also measured the overhead of preparing a masked DBMS to undergo validation and the overhead of bringing the masked DBMS back online after validation completes successfully. As discussed below, these overheads do not always affect the performance of the online processing of requests.

When using replicas, the time it takes for the operator to act on the masked DBMS affects the overhead of preparing the DBMS for validation, which is essentially the overhead of resynchronizing the online and masked DBMSs. For example, assuming the same infrastructure as the experiments above, 60 requests/second, and 10 minutes to complete the operator's task, we find that preparing a masked DBMS to undergo replica-based validation takes 51 seconds. For a task taking 20 minutes, preparing the masked DBMS takes 98 seconds. These overheads are directly related to the percentage of requests that induce database writes in our workload (6%). During resynchronization, neither the average throughput nor the average latency of online requests is noticeably affected. However, resynchronization does incur an additional 24–31% of average CPU utilization on the controller. After a successful replica-based validation, reintegration of the masked DBMS takes only milliseconds, since the two DBMSs are already synchronized.

When using traces, the overhead of preparing a masked DBMS for validation is not affected by the length of the operator task. Rather, this overhead is dominated by the time it takes to initialize the masked DBMS with the database state stored in the trace. For our 4GB auction database, this process takes 122 seconds. This overhead has no effect on the online requests, since it is only incurred by the masked DBMS. After a success-

ful trace-based validation, the overhead of reintegration is dominated by the resynchronization with the online DBMS. Resynchronization time essentially depends on how long trace replay lasts, leading to similar overheads to preparing a masked DBMS for replica-based validation. For example, if replay lasts 10 minutes, reintegration takes around 51 seconds. The impact of resynchronization on the processing of online requests is also the same as in replica-based validation.

Summary. Overall, these results are quite encouraging since the overheads we observed only impact the C-JDBC controller and only while validation is taking place. Furthermore, services typically run at mid-range resource utilizations (e.g., 50%–60%) to be able to deal with load spikes, meaning that the CPU overhead of validation should not affect throughputs in practice. Operating on the database during periods of low load reduces the potential impact of validation even further.

6 Discussion

In this section, we draw several interesting observations from our experience with DBA mistakes and database validation, as well as relate our findings to our previous validation work [16] on Web and application servers.

First, our survey clearly shows that most DBA mistakes are due to the separation and differences between online and testing environments. We believe that keeping these environments similar (ideally equal) is more difficult for database systems than for Web and application servers. The reason is that the amount of state that would need to be replicated across the environments can be orders of magnitude larger and more complex in the case of databases. This observation leads us to believe that deployment and performance mistakes will always be more common in database systems; structure mistakes have no clear equivalent in the context of Web and application servers. In contrast, configuration mistakes that are dominant in the latter systems are not so frequent in databases.

Second, it is clear also that DBA support tools can help database administration. However, these tools are very specific to DBMS and to the tasks that they support. We believe that validation (or a validation tool) is more generally applicable and thus potentially more useful. The negative side is that a validation tool on its own would not substantially reduce the amount of work required of the DBA; instead, it would simplify deployment and hide any mistakes that the DBA might make.

Third, we found that implementing validation for database systems is substantially more complex than doing so for Web and application servers. There are 3 reasons for the extra complexity: the amount of state in-

volved, the type of replication across DBMSs, and the consistency requirements of the state. The amount of state has implications on performance and request buffering, since requests need to be blocked during certain state management operations. Further, since database systems deal essentially with hard state, replication and state management have to maintain exact database replicas online. Related to the hard state, the strong consistency requirements of ACID force requests to be replayed in exactly the same order at the replicated databases. Strong consistency imposes extra constraints on how requests can be forwarded to (or replayed in) the validation slice. In contrast, Web and application servers involve relatively small amounts of soft state, do not require exact replication (functionality replication is enough), and only require the replicated ordering of the requests within each user session (rather than full strong consistency).

Fourth, we observe that making database structure changes and performing model-based validation without blocking any online requests is a challenge (one that C-JDBC does not address at all). The problem is that, during validation, the SQL commands that arrive in the online slice might actually explicitly refer to the old structure. When it is time to reintegrate the masked DBMS, any write commands that conflict with the new structure become incorrect. We did not face this problem in our previous work, as operations that would correctly change the behavior of the servers were not considered.

Finally, despite the above complexities, we believe that validation is conceptually simple to apply across different classes of systems. The key requirements are: (1) a component to be validated must have one or more replicas that are at least functionally equivalent to the component; (2) the system must be able to correctly adapt to component additions and removals; and (3) the system must support a means for creating a consistent snapshot of its state. For a system with these requirements, validation involves isolating slices, shunting requests and replies, implementing meaningful comparator functions, and managing state.

7 Conclusions

In this paper, we collected a large amount of data on the behavior of DBAs in the field through a survey of 51 experienced DBAs maintaining real databases. Based on the results of our survey, we proposed that a validation infrastructure that allows DBAs to check the correctness of their actions in an isolated slice of the online system itself would significantly reduce the impact of mistakes on database performance and availability. We designed and implemented a prototype of such a validation infrastructure for replicated databases. One novel aspect of this infrastructure is that it allows components of a repli-

cated database to be acted upon and validated while the database itself remains operational. We also proposed a novel validation strategy called model-based validation for checking the correctness of a component in the absence of any known correct instances whose behaviors can be used as a basis for validation. We showed how even a simple implementation of this strategy can be quite powerful in detecting DBA mistakes. We also showed that validation is quite effective at masking and detecting DBA mistakes; our validation infrastructure was able to mask 19 out of 23 injected mistakes, where the mistakes were designed to represent actual problems reported in our survey.

We now plan to explore model-based validation further, not only in the context of database systems but also for other systems, such as load balancers and firewalls.

Acknowledgments

We would like to thank Yuanyuan Zhou and the anonymous reviewers for comments that helped improve the paper. The work was partially supported by NSF grants #EIA-0103722, #EIA-9986046, #CCR-0100798, and #CSR-0509007.

References

- [1] BARHAM, P., ISAACS, R., MORTIER, R., AND NARAYANAN, D. Magpie: Real-Time Modelling and Performance-Aware Systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (May 2003).
- [2] BARRETT, R., MAGLIO, P. P., KANDOGAN, E., AND BAILEY, J. Usable Autonomic Computing Systems: The Administrator's Perspective. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)* (May 2004).
- [3] BROWN, A. B. *A Recovery-oriented Approach to Dependable Services: Repairing Past Errors with System-wide Undo*. PhD thesis, Computer Science Division, University of California, Berkeley, 2003.
- [4] BROWN, A. B., AND PATTERSON, D. A. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the 2003 USENIX Annual Technical Conference* (June 2003).
- [5] CASTRO, M., AND LISKOV, B. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI'00)* (Oct. 2000).
- [6] CASTRO, M., AND LISKOV, B. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)* (Oct. 2001).

- [7] CECCHET, E., MARGUERITE, J., AND ZWAENEPOEL, W. C-JDBC: Flexible Database Clustering Middleware. In *Proceedings of the USENIX Annual Technical Conference, Freenix track* (June 2004).
- [8] CHEN, M. Y., ACCARDI, A., KICIMAN, E., LLOYD, J., PATTERSON, D., FOX, A., AND BREWER, E. Path-Based Failure and Evolution Management. In *Proceedings of the International Symposium on Networked Systems Design and Implementation (NSDI'04)* (Mar. 2004).
- [9] CHEON, Y., AND LEAVENS, G. T. A Runtime Assertion Checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02)* (June 2002).
- [10] GIL, P., ARLAT, J., MADEIRA, H., CROUZET, Y., JARBOUI, T., KANOUN, K., MARTEAU, T., DURES, J., VIEIRA, M., GIL, D., BARAZA, J.-C., AND GRACIA, J. Fault Representativeness. Technical Report IST-2000-25425, Information Society Technologies, June 2002.
- [11] GRAY, J. Why do Computers Stop and What Can Be Done About It? In *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems* (Jan. 1986).
- [12] GRAY, J. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability* 39, 4 (Oct. 1990).
- [13] KALBARCZYK, Z. T., IYER, R. K., BAGCHI, S., AND WHISNANT, K. Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Transactions on Parallel and Distributed Systems* 10, 6 (1999).
- [14] LI, X., MARTIN, R. P., NAGARAJA, K., NGUYEN, T. D., AND ZHANG, B. Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services. In *Proceedings of 1st Workshop on Novel Uses of System Area Networks(SAN-1)* (Jan. 2002).
- [15] MULLINS, C. S. *Database Administration: The Complete Guide to Practices and Procedures*. Addison-Wesley, 2002.
- [16] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004).
- [17] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. Why do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Mar. 2003).
- [18] PERL, S. E., AND WEIHL, W. E. Performance Assertion Checking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Dec. 1993).
- [19] RICE UNIVERSITY. DynaServer Project. <http://www.cs.rice.edu/CS/Systems/DynaServer>, 2003.
- [20] VIEIRA, M., AND MADEIRA, H. A Dependability Benchmark for OLTP Application Environments. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)* (Sept. 2003).
- [21] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using Model Checking to Find Serious File System Errors. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004).

SMART: An Integrated Multi-Action Advisor for Storage Systems

Li Yin[†] Sandeep Uttamchandani[‡] Madhukar Korupolu[‡] Kaladhar Voruganti[‡] Randy Katz[†]
[†] University of California, Berkeley [‡] IBM Almaden Research Center

Abstract

The designers of clustered file systems, storage resource management software and storage virtualization devices are trying to provide the necessary planning functionality in their products to facilitate the invocation of the appropriate corrective actions in order to satisfy user specified service level objectives (SLOs). However, most existing approaches only perform planning for a single type of action such as workload throttling, or data migration, or addition of new resources. As will be shown in this paper, single action based plans are not always cost effective. In this paper we present a framework SMART that considers multiple types of corrective actions in an integrated manner and generates a combined corrective action schedule. Furthermore, often times, the best cost-effective schedule for a one-week lookahead could be different from the best cost-effective schedule for a one-year lookahead. An advantage of the SMART framework is that it considers this lookahead time window in coming up with its corrective action schedules. Finally, another key advantage of this framework is that it has a built-in mechanism to handle unexpected surges in workloads. We have implemented our framework and algorithm as part of a clustered file system and performed various experiments to show the benefits of our approach.

1 Introduction

With an increase in the number of applications, the amount of managed storage, and the number of policies and best practices, system administrators and designers are finding it extremely difficult to generate cost effective plans that can satisfy the storage needs of the storage applications. Typically, system administrators over provision their storage resources due to lack of proper storage management tools. Thus, the vendors of file systems, storage resource management software and storage virtualization boxes are trying to have the ability to

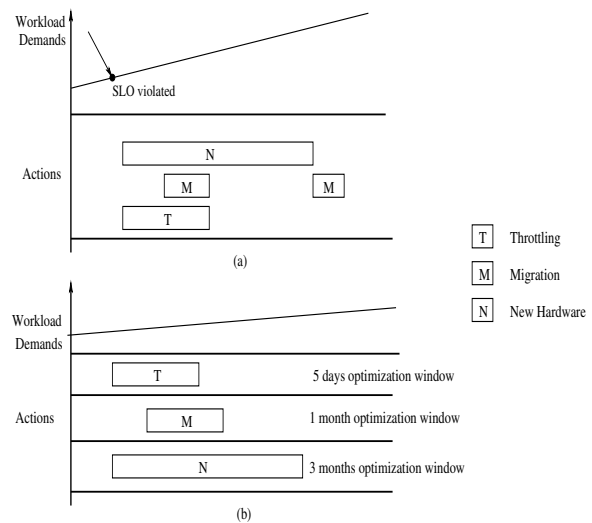


Figure 1: Planning Issues

automatically *monitor* resource utilization and workload SLOs, *analyze* the source of the problem, *plan* a corrective action, and *invoke* corrective actions to make the necessary changes.

Research prototypes and products that perform the above tasks are beginning to appear. However, these tools suffer from the following two key drawbacks with respect to the *planning* aspect of the solution:

Single Action Based: Existing effort has focused on using a single type of corrective action such as workload throttling, data migration, or adding more resources to correct SLO violations. These tools lack the ability to combine the different types of corrective actions to provide better cost trade-off points. When the SLO for a workload gets violated there will be situations where a combination of actions would provide the most optimum cost savings. For example, in Figure 1 (a), upon the violation of the SLO, it is desirable to throttle the workload until the data associated with the workload can be migrated to a less contended array rank, and if

necessary can be eventually migrated to a new rank that is part of a newly provisioned array. Thus, there is a need for a planning tool that can advise the administrators to take the right action or a combination of actions at the right time. Currently, the storage management eco-systems from various vendors provide good domain specific planning tools such as network planning, storage controller planning, migration planning etc. However, most of these tools are not integrated with each other. For example, capacity planning (typically considered a long term action) is not well integrated with throttling planning (an instantaneous action). Lack of proper integration between the planning tools transfers the responsibility of integration to a system administrator. As discussed above, this becomes difficult as the system size scales up. Typically this results in solutions that are either grossly over-provisioned with excess capacity or under-provisioned to meet service level agreements (SLOs).

Single time-window based optimization: Another drawback of existing tools is that they take a “one-size fits all” approach to the problem. For example, the solution (workload throttling) that is the most cost effective for one week might be different from the solution (adding new hardware) that is the most optimum for one year. Time is an important aspect and often overlooked part of the planning process. That is, cost-wise different solutions could be optimal during different observation windows. Currently, most storage planning tools do not allow administrators to evaluate plans for different observation time windows. This results in the administrators not taking the right action at the right time. For example, as shown in Figure 1 (b), different solutions are optimal for different time windows: (1) If the optimization time window is five days, throttling is the most optimal solution. (2) If the optimization time window is one month, data migration is the most cost effective solution. (3) If the optimization time window is three months, addition of new hardware is the most cost effective solution.

In this paper, we propose an action schedule framework called SMART. The key contributions of SMART are:

Integrated multi-action planning: We provide an action scheduling algorithm that allows combining seemingly disparate storage management actions such as workload throttling, data migration, and resource provisioning into an integrated framework.

Multi-granularity temporal planning: Our algorithm allows for the specification of optimization time windows. For example, one could indicate that they want the solution that is the most cost effective for either one day or one year.

Action selection for unexpected workload variations:

Our core algorithm (contribution number 1) can determine whether the surge in the I/O requests is an unknown workload spike or a known trend and select corrective actions accordingly.

Deployment of the framework in a file-system: In order to validate the benefits described above, we have implemented our framework and algorithm as part of the GPFS: a scalable shared-disk file system [21]. GPFS performs its own logical volume management. We have evaluated our implementation and the results are presented in the experiment section. It is to be noted that the framework and algorithm are general enough to be deployed as part of storage resource management and storage virtualization software.

2 Framework for SMART

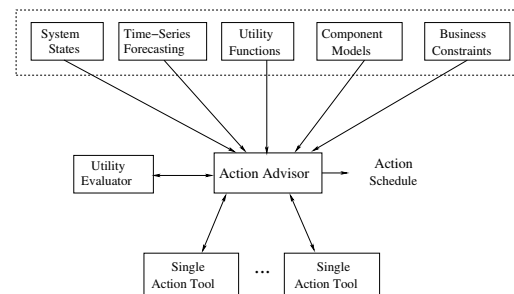


Figure 2: Architecture of Action Scheduler

This section describes the framework of SMART (shown in Figure 2). SMART can be deployed in file systems, storage resource management software and storage virtualization boxes (details of the file system deployment are given in Section 4). The key components of SMART are:

Input modules: They include sensors monitoring the system state \mathcal{S} , specifications for administrator-defined business-level constraints (budget constraints and optimization window), SLOs, utility functions, time-series forecasting of workload request-rate, and component models for the storage devices. Here, the system state represents the run-time details of the system and is defined as a triplet $\mathcal{S} = \langle \mathcal{C}, \mathcal{W}, \mathcal{M} \rangle$, where \mathcal{C} is the set of components in the system, \mathcal{W} is the workloads and \mathcal{M} is the current mapping of workloads to the components.

Utility evaluator: It calculates the overall utility value in a given system state. It uses component models to interpolate the IO performance values which in turn map to the utility delivered to the workloads.

Single action tools: They decide the optimal invocation parameters for a corrective action in a given system state. SMART can leverage existing tools for individual corrective actions namely throttling [25, 9, 17], migration [10, 16], and provisioning [5, 3].

Action Advisor: This is the **core** of SMART that aims to improve storage system utility for a given optimization window and business-level constraints. It interacts with the single action tools and generates a time-based action schedule with details of what action to invoke, when to invoke and how to invoke. This is accomplished by feeding the action tools with different system states and collecting individual action options. The Action Advisor then analyzes the selected action invocation parameters using the Utility Evaluator (details of the algorithm are presented in Section 3). The Action Advisor can operate both reactively (when SLO has been violated) as well as proactively (before SLO violation happens).

This section covers the details of the framework. We will present the details of the algorithm for the Action Advisor in the next section.

2.1 Input modules

For the input modules described below, there are several different techniques that are available – the focus of this paper is to demonstrate how these building blocks work together to solve the problem, rather than their internal details.

Time-series Forecasting

The forecasting of future workload demands is based on extracting patterns and trends from historical data. There are several well-known approaches for time series analysis of historic data such as ARIMA [24] and Neural Network [8]. The general form of time-series functions is as follows:

$$y_{t+h} = g(X_t, \theta) + \epsilon_{t+h} \quad (1)$$

where: y_t is the variable(s) vector to be forecast. t is the time when the forecast is made. X_t are predictor variables, which usually includes the observed and lagged values of y_t till time t . θ is the vector of parameter of the function g and ϵ_{t+h} is the prediction error.

Utility functions

The concept of *utility function* has been introduced to evaluate the degree of user's satisfaction. There are several different techniques to specify utility functions. For SMART, the utility function associates workloads performance with a utility value, which reflects the user's degree of satisfaction. The utility function for each workload can be (1) provided by the administrators; (2) defined in terms of priority value and SLOs; or (3) defined by associating a dollar value to the level of service delivered, e.g., \$1000/GB if the latency is less than 10ms, otherwise \$100/GB.

Component models

A component model predicts values of a delivery metric

as a function of workload characteristics. SMART can in principle accommodate models for any system component. In particular, the model for a storage device takes the form:

$$\text{Response_time} = c(\text{req_size}, \text{req_rate}, \text{rw_ratio}, \text{random/sequential}, \text{cache_hit_rate})$$

Creating component models is an area of active ongoing research. Models based on simulation or emulation [12, 29] require a fairly detailed knowledge of the system's internals; analytical models [22, 19] require less, but device-specific information must still be taken into account to obtain accurate predictions. *Black-box* [4, 26] models are built by recording and correlating inputs and outputs to the system in diverse states, without regarding its internal structure. Since SMART needs to explore a large candidate space in a short time, simulation based approaches are not feasible due to the long prediction overhead. Analytical models and black box approaches both work with SMART. For the SMART prototype, we use a regression based approach to bootstrap the models and refine models continuously at run time.

2.2 Utility Evaluator

As the name suggests, the *Utility Evaluator* calculates the overall utility delivered by the storage system in a given system state. The calculation involves getting the access characteristics of each workload, and using the component models to interpolate the average response-time of each workload. The Utility Evaluator uses the throughput and response-time for each workload to calculate the utility value delivered by the storage system:

$$U_{sys} = \sum_{j=1}^N UF_j(\text{Thru}_j, \text{Lat}_j) \quad (2)$$

where N is the total number of workloads, UF_j is the utility function of workload j , with throughput Thru_j and latency Lat_j .

In addition, for any given workload demands D_j , the system maximum utility value $UMax_{sys}$ is defined as the “ideal” maximum utility value if the requests for all workloads are satisfied. Utility loss UL_{sys} is the difference between the maximum utility value and the current system utility value. They can be calculated as follows:

$$\begin{aligned} UMax_{sys} &= \sum_{j=1}^N UF_j(D_j, SLO_{lat_j}) \\ UL_{sys} &= UMax_{sys} - U_{sys} \end{aligned} \quad (3)$$

where the SLO_{lat_j} is the latency requirement of workload j . In addition, cumulative utility value for a given time window refers to the sum of the utility value across the time window.

2.3 Single Action Tools

These tools automate invocation of a single action. A few examples are Chameleon [25], Facade [17] for throttling; QoS Mig [10], Aqueduct [16] for migration; Ergastulum [6], Hippodrome [5] for provisioning. Each of these tools typically includes the logic for deciding the action invocation parameter values, and an executor to enforce these parameters.

The single action tools take the system state, performance models and utility functions as input from the Action Advisor and outputs the invocation parameters. For example, in the case of migration, it decides the data to be migrated, the target location, and the migration speed. Every action has a cost in terms of the resource or budget overhead and a benefit in terms of the improvement in the performance of the workloads. The action invocation parameters are used to determine the resulting performance of each workload and the corresponding utility value.

2.4 Action Advisor

The Action Advisor generates the corrective action schedule – the steps involved are as follows (details of the algorithm are covered in the next section):

- Generate and analyze the current state (S_0) as well as lookahead states (S_1, S_2, \dots) according to the forecasted future.
- Feed the system states along with the workload utility functions and performance models to the single action tools and collect their invocation options
- Analyze the cost-benefit of the action invocation options – this is accomplished using the Utility Evaluator module.
- Prune the solution space and generate a schedule of what actions to invoke, when to invoke, and how to invoke.

3 Algorithm for Action Advisor in SMART

Action Advisor is the core of SMART: it determines an action schedule consisting of one or more actions (*what*) with action invocation time (*when*) and invocation parameters (*how*). The goal of Action Advisor is to pick a combination of actions that will improve the overall system utility or, equivalently, reduce the system utility loss. In the rest of this section, we will first intuitively motivate the algorithm and give the details after that.

The Action Advisor operates in two different modes depending on whether the corrective actions are being invoked proactively in response to forecasted workload growth, or reactively in response to unexpected variations in the workloads. The former is referred to as the *normal mode*, while the later is the *unexpected mode*.

In the *normal mode*, SMART uses an approach similar to the *divide-and-conquer* concept. It breaks the optimization window into smaller unequal sub-windows and uses a *recursive greedy with look-back and look-forward* approach to select actions within each sub-window. The motivation of divide-and-conquer is to reduce the problem complexity and to treat the near-term fine-grained prediction periods differently from the long-term coarse-grained prediction periods. The action selection for each sub-window is performed in a sequential fashion, i.e., the resulting system state of one sub-window acts as the starting state for the next consecutive window. The divide-and-conquer approach reduces the problem complexity at the cost of optimality: the sum of local optimal actions for each sub-window may not lead to the global optimal.

The *unexpected mode* selects actions defensively. It tries to avoid invoking expensive actions since the workload variation could go away soon after the action is invoked, making the overhead wasted. However this needs to be balanced with the potential risk of the high workload persisting and thus incurring continuous utility loss which may add up over time. We formulate this analysis as a decision-making problem with unknown future and apply the “ski-rental” online algorithm [15] to select actions.

Action Advisor uses a primitive mechanism to transition between normal and unexpected modes. It continuously compares the observed values against the predicted values (using any chosen predictor model, for example, ARIMA). If the difference is large then it moves into the defensive unexpected workload mode. While in that mode, it continuously updates its predictor function based on the newly observed values. When the predicted values and observed values are close enough for a sufficiently long period, it transitions to the normal mode.

In the rest of this section, we first present action selections for *normal mode* and *unexpected mode* respectively. After that, we discuss the *risk modulation* which deals with the future uncertainty and action invocation overhead.

3.1 Normal Mode: Greedy Pruning with Look-back and Look-forward

The Action Advisor takes as input the current system state, the workload prediction, the utility functions, the available budget B for new hardware and the length of the optimization window. The action selection begins by breaking the specified optimization window into smaller unequal sub-windows. For instance, a one year optimization window is split into sub-windows of 1 day, 1 month, and 1 year; The number and length of the sub-windows can be configured by the administrator. Within each sub-

window $[T_k, T_{k+1}]$, the goal is to find actions that maximize the cumulative system utility in the sub-window. This process is formulated as a tree-construction algorithm (Figure 3) as described below.

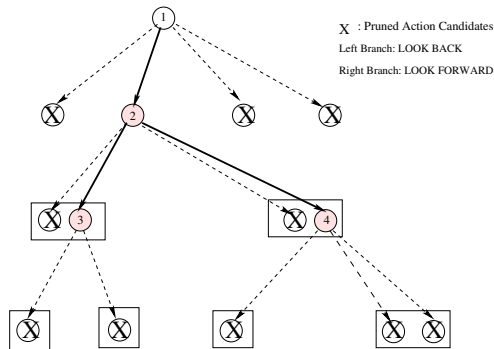


Figure 3: Tree Based Action Schedule Generation

In the tree-based representation, the root corresponds to the entire sub-window, $[T_k, T_{k+1}]$. The branches originating from the root represent the candidate actions returned by single action tools. For m possible action options there will be m branches. The resulting node i for each action has the following information:

- The selected action and its invocation parameters.
- The action invocation time and finish time $[invoke_i, finish_i]$.
- The decision window start and end time $[start_i, end_i]$. For nodes originating from the root, the value is $[T_k, T_{k+1}]$.
- The initial S_i and resulting state S_{i+1} .
- The predicted cumulative utility loss UL_i , defined as the sum of system utility loss from $start_i$ to end_i if action i is invoked.

Greedy Pruning: Using the basic greedy approach, the Action Advisor selects a first-level node in the tree that has the lowest utility loss UL_i and prunes the other $m - 1$ branches (circles crossed out in Figure 3). In addition, a threshold is introduced to ensure that the action gives sufficient improvement. The selected action will only be scheduled if the improvement exceeds the threshold. The threshold is configurable such that a higher value leads to more aggressive pruning. This greedy pruning procedure is referred to as function *GreedyPrune* in the pseudocode described later.

Lookback and Lookforward Optimization: In real-world systems, it may be required to invoke more than one action concurrently. For example, if data migration is selected, it might be required to additionally throttle the lower priority workloads until all data are migrated. The Action Advisor uses the *Look-back* and

Look-forward Optimization to improve the action plan. The look-back and look-forward are with reference to the selected action's finish time $finish_i$. Look-back seeks action options in the time window $[start_i, finish_i]$ (before the selected action finishes). Look-forward examines possible actions in the window $[finish_i, end_i]$ (after the selected action finishes). Time $finish_i$ is chosen as the splitting point because (1) the system state is permanently changed after the action finishes, making the cost-benefit of action options changed and (2) any action scheduled before the selected action finishes need to satisfy the no-conflict constraint (described later). Essentially, the look-back and look-forward optimization splits the time window recursively and seeks actions to improve the system utility further. In the tree-construction, the action candidates for look-back and look-forward are represented as left and right children (marked as solid circles in Figure 3). The *pruning-lookback-lookforward* procedure is recursively performed to construct an action schedule until the *GreedyPrune* finds no action option. The pseudocode for look-forward and look-back optimization is given in function *Lookback* and *Lookforward* respectively.

```

Function Lookback(i) {
    Foreach (Corrective_actions) {
        If (!(Conflict(Existing_actions)) {
            Find action option in (start_i, finish_i);
            Add to Left_Children(i);
        }
    }
    GreedyPrune(Left_Children(i));
    If (Left_Children(i) != NULL) {
        Lookback(Left_Children(i));
        Lookforward(Left_Children(i));
    }
}

Function Lookforward(i) {
    Foreach (Corrective_actions) {
        Find action option in (finish_i, end_i);
        Add to Right_Children(i);
    }
    GreedyPrune(Right_Children(i));
    If (Right_Children(i) != NULL) {
        Lookback(Right_Children(i));
        Lookforward(Right_Children(i));
    }
}

```

When considering actions to be scheduled before another action finishes (in lookback phase), the actions should not conflict with existing selected actions. Two actions conflict if one of the following is true:

- If they depend on the same resource.
- If action j overlaps with an action k already in the schedule, and action j violates the precondition for action k . For example, migration action 1 of moving data A from LUN_1 to LUN_2 will invalidate ac-

tion 2 of moving data A from LUN_1 to LUN_3 because the pre-condition of action 2 that data A was on LUN_1 is no longer true.

In summary, the Action Advisor generates the schedule of corrective actions using a recursive approach (the pseudocode is given below). The final action schedule for each sub-window is obtained by sorting the unpruned nodes (solid circles in Figure 3) in the tree according to their action invocation time ($invoke_i$).

```

Function TreeSchedule() {
    Foreach (Corrective_actions) {
        Find action option in  $[T_k, T_{k+1}]$ ;
        Add to Children(root);
    }
    GreedyPrune(Children(root));
    If (Children(root) != NULL) {
        Lookback(Children(root));
        Lookforward(Children(root));
    }
}

```

Finally, each sub-window is processed sequentially, i.e., the resulting system state of sub-window $[T_k, T_{k+1}]$ is the starting state of sub-window $[T_{k+1}, T_{k+2}]$, and the action schedules are composed into the final action schedule according to the action invocation time.

3.2 Unexpected Mode: Defensive Action Selection

Optimizing for the unexpected workload mode is challenging since it is difficult to predict the duration for which workload variation will persist. The Action Advisor uses a strategy similar to the one used in on-line decision making scenarios such as the “ski rental: to rent or to buy” [15]. There the choice of whether to buy (at cost say \$150) or rent a ski (at cost say \$10 per trip) has to be made without the knowledge of the future (how many times one might go skiing). If one skis less than 15 times, then renting is better, otherwise buying is better. In the absence of the knowledge of the future, the commonly used strategy is “to keep renting until the amount paid in renting equals the cost of buying, and then buy.” This strategy is always within a factor of two of the optimal, regardless of how many times one goes skiing, and is probably the best possible in the absence of knowledge about the future.

The Action Advisor follows a similar online strategy. It selects the least costly action until the cumulative utility loss for staying with that action exceeds the cost of invoking the next expensive action. When SMART is in the *unexpected* mode, the Action Advisor first finds all action candidates under the assumption that the system state and workload demands will remain the same. For

each candidate A_i , the cost is initialized as the extra utility loss and hardware cost (if any) paid for the action invocation (shown in Equation 4):

$$Cost(A_i) = \sum_{t=0}^{leadtime(A_i)} (U_{sys}(no_action, t) - U_{sys}(A_i_ongoing, t)) + HW_Cost(A_i) \quad (4)$$

Where $U_{sys}(noaction, t)$ is the system utility value at time t if no corrective action is taken and $U_{sys}(A_i_ongoing, t)$ is the system utility at time t if A_i is ongoing. For example, for throttling, $Cost(A_i)$ will be zero because the leadtime is zero. For migration, the $Cost(A_i)$ is the total utility loss over $leadtime(A_i)$ due to allocating resources to move data around.

Action Advisor selects the action with minimum cost and invokes it immediately. Over time, the cost of each action candidate (including both the selected one and unchosen ones) is updated continuously to reflect the utility loss experienced if A_i had been invoked. Equation (5) gives the value of $Cost(A_i)$ after t intervals:

$$Cost(A_i) = Cost(A_i) + \sum_{j=0}^t UL(A_i, j) \quad (5)$$

This cost updating procedure continues until following situations happen:

- Another action k has a lower cost than the previously invoked action. Action Advisor invokes action k immediately and continues the cost updating procedure. For example, if the system experiences utility loss with throttling, but has no utility loss after migration, the cost for throttling action will continuously grow and the cost of migration will stay same over time. At some point, the cost of throttling will exceed the cost of migration and the migration option will be invoked by then.
- System goes back to a good state for a period of time. The Action Advisor will stop the action selection procedure (exception has gone).
- The system collects enough new observations and transitions back to *normal* mode.

3.3 Risk Modulation

In our previous discussion, action selection has been made based on the cumulative utility loss UL_i . The accuracy of UL_i depends on the accuracy of future workload forecasting, performance prediction and cost-benefit effect estimation of actions. Inaccurate estimation of UL_i may result in decisions leading to reduced overall utility. To account for the impact of inaccurate input information, we perform risk modulation on the UL_i

for each action option. Here, *risk* captures both the probability that the utility gain of an action will be lost (in the future system-states) as a result of volatility in the workload time-series functions (e.g., the demand for W_1 was expected to be 10K IOPS after 1 month, but it turns out to be 5K, making the utility improvement of buying new hardware wasted) and the impact of making a wrong action decision (e.g., the impact of a wrong decision to migrate data when the system is 90% utilized is higher than that of when the system is 20% loaded).

There are several techniques for measuring risk. Actions for assigning storage resources among workloads are analogous to portfolio management in which funds are allocated to various company stocks. In economics and finance, the *Value at Risk* (VaR) [11] is a technique used to estimate the probability of portfolio losses based on the statistical analysis of historical price trends and volatilities in trend prediction. In the context of SMART, *VaR* represents the probability with a 95% confidence, that the workload system will not grow in the future, making the action invocation unnecessary.

$$VaR(95\% \text{ confidence}) = -1.65\sigma \times \sqrt{T} \quad (6)$$

where, σ is the standard deviation of the time-series request-rate predictions and T is the number of days in the future for which the risk estimate holds. For different sub-windows, the prediction standard deviation may be different: a near-term prediction is likely to be more precise than a long-term one.

The risk value $RF(A_i)$ of action i is calculated by:

$$RF(A_i) = -(1 + \alpha) * VaR \quad (7)$$

where α reflects the risk factors of an individual action (based on its operational semantics) and is defined as follows:

$$\begin{aligned} \alpha_{thr} &= 0 \\ \alpha_{mig} &= \frac{bytes_moved}{total_bytes_on_source} * Sys_Utilization \\ \alpha_{hw} &= \frac{hardware_cost}{total_budget} * (1 - Sys_Utilization) \end{aligned}$$

Where *SysUtilization* is the system utilization when the action is invoked.

For each action option returned by single action tools, the Action Advisor calculates the risk factor $RF(A_i)$ and scales the cumulative utility loss UL_i according to Equation 8 and the action selection is performed based on the scaled UL_i^* (For example, in GreedyPrune).

$$UL_i^* = (1 + RF(A_i)) \times UL_i \quad (8)$$

4 Experiments

SMART generates an action schedule to improve system utility. To evaluate the quality of its decision, we imple-

mented SMART in both a real file system GPFS [21] and a simulator. System implementation allows us to verify if SMART can be applied practically while simulator provides us a more controlled and scalable environment, which allows us to perform repeatable experiments to gain insights on the overhead and sensitivity to input information errors.

The experiments are divided into three parts: First, *sanity check* experiments are performed to examine the impact of various configuration parameters on SMART's decision. Secondly, *feasibility experiments* evaluate the behavior of two representative cases in the *sanity check* using the GPFS prototype. Third, *sensitivity test* first examines the quality of the decisions with accurate component models and future prediction over a variety of scenarios using simulator. It then varies the error rate of the component models and time series prediction respectively and evaluates their impact on SMART's quality. In addition, SMART is designed to assist the administrators to make decisions. However, in order to examine the quality of SMART's decision (for example, what will happen if the administrator follows SMART's advice), selected actions are automatically executed in the experiments.

In the rest of this section, we first describe our GPFS prototype implementation and then present the experimental results of three tests.

4.1 GPFS Prototype Implementation

The SMART prototype is implemented on GPFS: a commercial high-performance distributed file-system [21]. GPFS manages the underlying storage systems as *pools* that differ in their characteristics of capacity, performance and availability. The storage systems can be accessed by any clients nodes running on separate physical machines transparently.

The prototype implementation involved *sensors* for monitoring the workloads states, *actuators* for executing corrective actions and an Action Advisor for decision making.

Sensors: They collect information about the run-time state of workloads. The monitor daemon in each GPFS client node tracks the access characteristics of the workload and writes it to a file, which can be analyzed periodically in a centralized fashion. Workloads are the unit of tracking and control – in the prototype implementation, a workload is defined manually as a collection of PIDs assigned by the OS. The monitoring daemon does book-keeping at the GPFS read/write function call invoked after the VFS translation.

Action actuators: Although the long term goal of the prototype is to support all corrective actions, as a proof of

concept, we first implemented action actuators for three most commonly used corrective actions: throttling, migration and adding new pools.

- The IO throttling is enforced at the GPFS client nodes using a token-bucket algorithm. The decision-making for throttling each workload is made in a centralized fashion, with the token-issue rate and bucket size written to a control file that is then periodically (20ms) checked by the node throttling daemon.
- Similarly, the control file for the migration daemon consists of entries of the form <file name, source pool, the destination pool> and the migration speed is controlled by throttling the migration process. The migration daemon thread runs in the context of one of the client nodes and periodically checks for updates in the control file and invokes the GPFS built in function `mmchattr` to migrate files.
- Because the addition of hardware normally requires human intervention, we mimic the effect of adding new hardware by pre-reserving storage devices and forbidding the access to them until SMART decides to add them into the system. In addition, the storage devices are configured with different leadtime to mimic the overhead of placing orders and installation.

In addition, the daemon threads for throttling and migration run at the user-level and invoke kernel-level ioctl for the respective actions.

Action Advisor Integration: The SMART action advisor is implemented using techniques described in Section 3. The time-series forecasting is done in an off-line fashion, where the monitored access characteristics for each workload are periodically fed to the ARIMA module [24] for refining the future forecast. Similarly, the performance prediction is done by bootstrapping the system for the initial models and refining the models as more data are collected. Once the Action Advisor is invoked, it communicates with individual action decision-making boxes and generates an action schedule. The selected actions are on hold by the action advisor until the action invocation time (determined by SMART) is due. At that time, the control files for corresponding action actuators are updated according to SMART's decision.

Without access to commercial decision tools, we implemented our own throttling, migration and provisioning tools. Throttling uses simulated annealing algorithm [20] to allocate tokens for each workload; the migration plan is generated by combining optimization, planning and risk modulation. It decides *what* and *where* to migrate, *when* to start migration and *migration speed*. The

provisioning decision is done by estimating the overall system utility for each provisioning option, which considers the utility loss before the hardware arrives, introduced by the load balancing operation (after the new hardware comes into place) and the financial cost of buying and maintaining the hardware.

4.2 Sanity Check

As a sanity check, the action advisor in the GPFS prototype is given the initial system settings as input, while the configuration parameters are varied to examine their impact on SMART's action schedule. The initial system setting for the tests is as follows:

Workload	Request size [KB]	Rd/wrt ratio	Seq/rnd ratio	Foot-print [GB]	ON/OFF phase [Hour]	ON/OFF [Iops]
W_{Trend}	16	0.7	0.8	60	12/12	150/100
W_{Backup}	16	1	1	600	8/16	250/0
W_{Phase}	8	0.8	0.9	6	14/10	150/100

Table 1: Access characteristics of workloads

Workloads: There are four workload streams: one is a 2 month trace replay of HP's Cello99 traces [18]. The other three workloads are synthetic workload traces with the following characteristics: 1) W_{Trend} is a workload with a growth trend – the ON phase load increases by 100 IOPS each day while the OFF phase load increases by 50 IOPS; 2) W_{Phase} is a workload with periodic ON-OFF phases; 3) W_{Backup} simulates a backup load with its ON-phase as an inverse of the *phased* workload. The access characteristics of these workloads are summarized in Table 1. Figure 5 (a) shows the IO rate of these workloads as a function of time. The default utility function for violating and meeting the SLO latency goals are shown in Figure 4 unless specified otherwise.

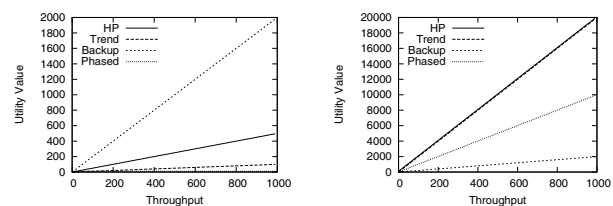


Figure 4: (a) Utility functions for violating SLO latency goals (b) Utility functions for meeting SLO latency goals

Components: There are three logical volumes: POOL1 and POOL2 are both RAID 5 arrays with 16 drives each, while POOL3 is a RAID 0 with 8 drives. POOL3 is originally off-line, and is accessible only when SMART selects hardware provisioning as a corrective action. The initial workload-to-component mapping is: [HP: POOL1], [Trend: POOL1], [Phased: POOL2] and [Backup: POOL2].

Miscellaneous settings: The optimization window is set

to one month; the default budget constraint is \$20,000 and the one day standard deviation of the load for risk analysis is configured as 10% unless otherwise specified. The provisioning tool is configured with 5 options, each with different buying cost, leadtime and estimated performance models. For these initial system settings, the system utility loss at different time intervals without any corrective action is shown in Figure 5 (b).

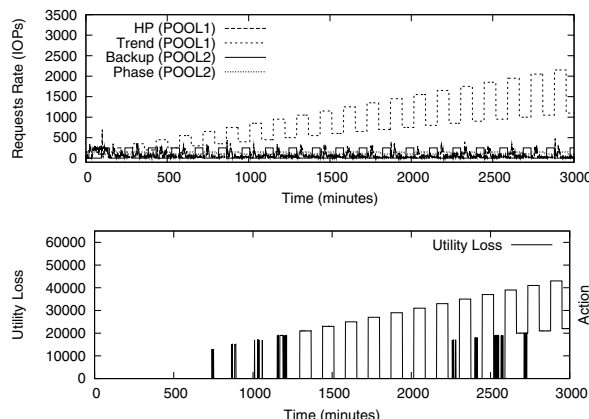


Figure 5: (a) Workload Demands (b) utility loss if no corrective action is invoked

An ideal yardstick to evaluate the quality of SMART's decisions is by comparing it with existing automated algorithms or with decisions made by an administrator. However, we are not aware of any existing work that considers multiple actions; also, it is very difficult to quantify decisions of a representative administrator. Because of this, we take an alternative approach of comparing the impact of SMART's decisions with the maximum theoretical system utility (upper bound, Equation 3) and the system utility without any action (lower bound).

In the rest of this section, using the system settings described above, we vary the configuration parameters that affect SMART's scheduling decisions namely the utility function values (test 1); the length of optimization window (test 2), budget constraints (test 3), risk factor (test 4). In test 5, we explore how SMART handles unexpected case. For each of these tests, we present the corrective action schedule generated by SMART, and the corresponding *predicted* utility loss as a function of time (depicted on the x-axis).

Test 1: Impact of Utility Function

SMART selects actions that maximize the overall system utility value, which is driven by the utility functions for individual workloads using the storage system. In this test, we vary W_{Trend} 's utility function of meeting the SLO latency goal from the default $20 * Thru$ to $540 * \log(Thru + 1)$. As shown in Figure 6(a), the default utility assignment for W_{Trend} causes a fast growing overall utility loss – SMART selects to add new hard-

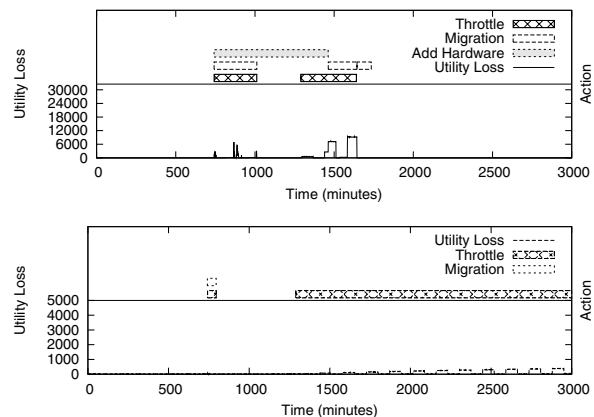


Figure 6: Action Invocation for different utility functions assigned to W_{trend} (a) Results for high utility value $UF_{trend} = 20 * Thru$ (default configuration) (b) Results for low utility value assignment $UF_{trend} = 540 * \log(Thru + 1)$

ware. However, for the low value utility assignment, the latency violation caused by the increasing load results in a much slower growth in the utility loss. As a result, the cost of adding new hardware cannot be justified for the current optimization window and hence SMART decides to settle for throttling and migration. Note that as the utility loss slowly approaches to the point where the cost of adding new hardware can be justified, SMART will suggest invoking hardware provisioning as needed.

Test 2: Impact of Optimization Window

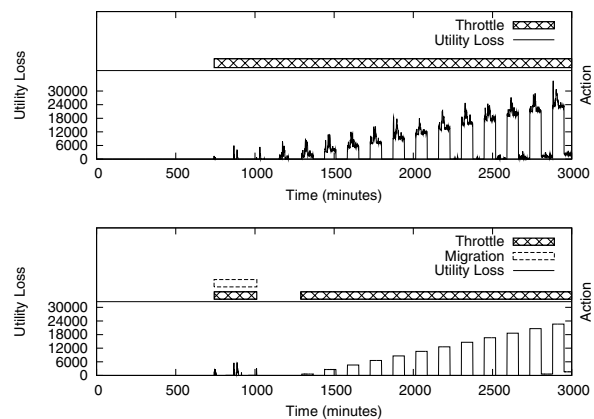


Figure 7: Action Invocation for different optimization windows (a) Results for 2 days optimization window (b) Results for 1 week optimization window

SMART is designed to select corrective actions that maximize the overall utility for a given optimization window. In this test, we vary the optimization window to 2 days, 1 week and 1 month (default value) – compared to the schedule for 1 month in Figure 6 (a), Figure 7 shows that SMART correctively chooses different action

schedules for the same starting system settings. In brief, for a short optimization window (Figure 7 (a) and (b)), SMART correctly selects action options with a lower cost, while for a longer optimization window (Figure 6 (a)), it suggests higher cost corrective options that are more beneficial in the long run.

Test 3: Impact of Budget Constraints

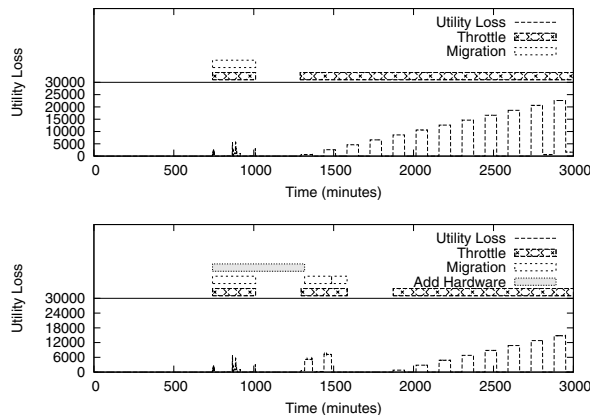


Figure 8: Action Invocation for different budget constraints (a) no budget available (b) low budget (\$5000)

Test 3 demonstrates how SMART responds to various budget constraints. As shown in Figure 8, SMART settles for throttling and migration if no budget is available for buying new hardware. With \$5,000 budget, SMART opts for adding a hardware. However, compared to the hardware selected for the default \$20,000 budget (shown in Figure 6 (a)), the hardware selected is not sufficient to solve the problem completely, and additionally requires a certain degree of traffic regulation.

Test 4: Impact of Risk Modulation

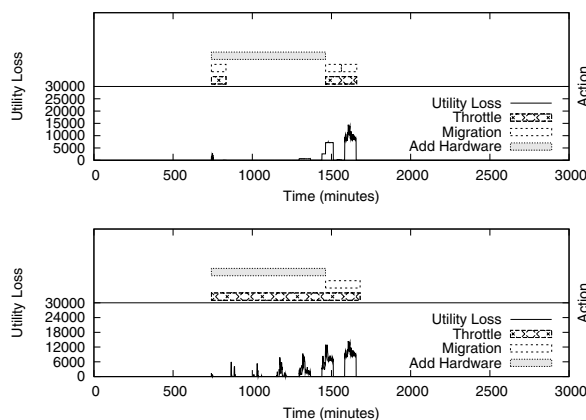


Figure 9: Action Invocation for different risk factor. (a) Available migration option involves 20GB data movement (b) Available migration involves 1000GB data movement

SMART uses risk modulation to balance between the risk of invoking an inappropriate action and the corresponding benefit on the utility value. For this experiment, the size of the dataset selected for migration is varied, changing the risk value (Equation 7) associated with the action options. SMART will select the high-risk option only if its benefit is proportionally higher. As shown in Figure 9, SMART changes the ranking of the corrective options and selects a different action invocation schedule for the two risk cases.

Test 5: Handling of Unexpected Case

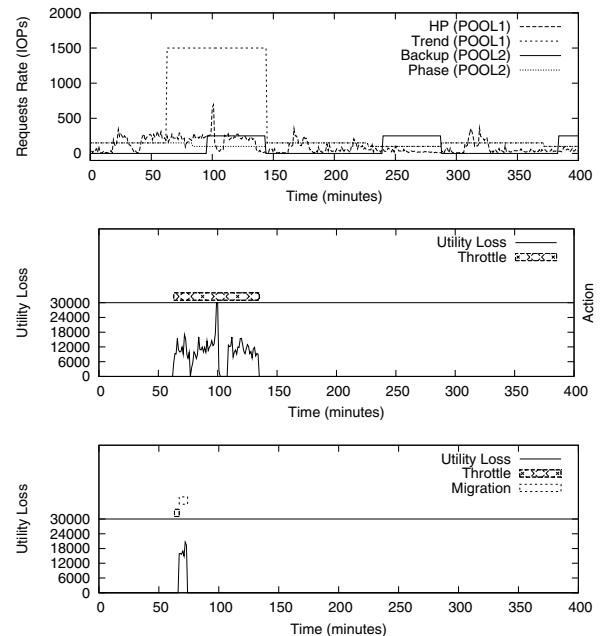


Figure 10: Action Invocation for unexpected case (a) Workloads Demands (b) Result for a “short” spike (c) results for a “long” spike

This test explores SMART’s ability to handle the unexpected workload demands. Figure 10 (a) shows the sending rate of the workload demands. From minute 60 to minute 145, W_{Trend} sends at 1,500 IOPS instead of the normal 250 IOPS. The difference between the predicted value 250 IOPS and the observed value 1,500 IOPS exceeds the threshold and SMART switches to unexpected mode. For both cases, SMART invokes throttling directly. But for case 1, the migration option involves a 1,000 GB data movement and is never invoked because the spike duration is not long enough to reach to a point where the migration invocation cost is less than the utility loss of staying with throttling. For case 2, a lower cost migration option is available (8GB data) and after 5 minutes, the utility loss due to settling for throttling already exceeds the invocation cost of migration. The migration option is invoked immediately as a result.

4.3 Feasibility Test Using GPFS Prototype

In these tests, SMART is run within the actual GPFS deployment. The tests serve two purposes: (1) to verify if the action schedule generated by SMART can actually help reduce system utility loss; (2) to examine if the utility loss predicted by SMART matches the observed utility loss. We run two tests to demonstrate the *normal mode* and *unexpected mode* operation.

Test 1: Normal Model

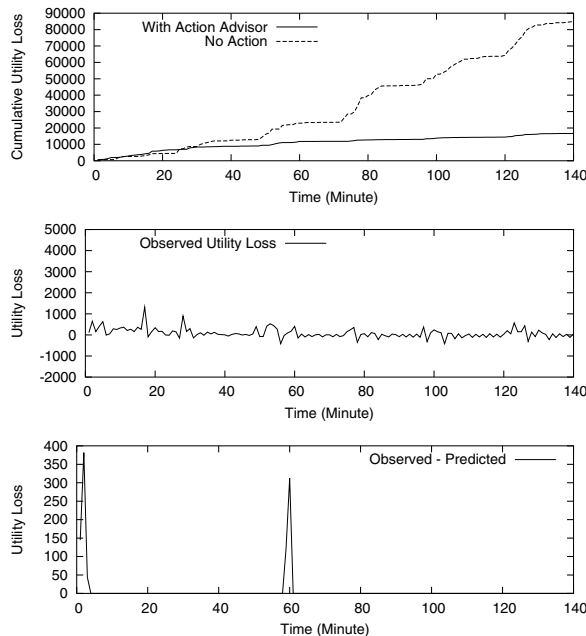


Figure 11: (a) cumulative utility loss comparison of without action and with SMART's decision (b) observed utility loss (c) difference in Utility Loss (after filtering): Observed value-Predicted

The setting for the experiment are the same as those used in the sanity check tests with two exceptions: the footprint size, and the leadtime of hardware addition. To reduce the experiment running time, the IO features are changed to run 60 times faster (every minute in the figure corresponds to one hour in the trace), and the footprint size is shrunk by a factor of 60, and the leadtime of adding hardware is set to 55 minutes (that maps to the original of 55 hours).

SMART evaluates the information and decides that the best option is to add a new pool. Because it takes 55 minutes to come into effect, SMART looks back to seek for solutions that can reduce the utility loss for time window [0, 55]. It chooses to migrate the *HP* workloads from POOL1 to POOL2 and throttle workloads until the new pool arrives. After POOL3 joins, the load balance operation decides to migrate the *Trend* workload to POOL3 and *HP* back to POOL1. The final workload to compo-

nent mapping is: *HP* on POOL1, *Backup* and *Phased* on POOL2 and *Trend* on POOL3.

As shown in Figure 11 (a), compared to without any action, SMART's action schedule eliminates about 80% of the original utility loss and also grows at a much slower rate. Before time 20, the no action utility loss is slightly lower than with SMART because the SMART schedule is paying extra utility loss for invoking the migration operation.

It can be observed from Figure 11 (b) that there is a negative utility loss. This is because the maximum utility is calculated based on the *planned* workload demands, while the observed utility value is calculated based on the *observed* throughput. Due to the lack of precise control in task scheduling, the workload generator can not precisely generate I/O requests as specified. For example: the workload generator for the HP traces is supposed to send out requests at a rate of 57.46 IOPS at time 33 while the observed throughput is actually 58.59 IOPS. As a result, the observed utility value is actually higher than the maximum value and results in a negative utility loss. For a similar reason, the observed utility loss fluctuates very frequently around zero utility loss.

SMART schedules actions and predicts the utility loss to be close to zero. However, the observed utility loss (shown in Figure 11 (b)) has non-zero values. In order to understand the cause of this, we filter out the amount of observed utility loss due to imprecise workload generation (described above), and plot the remaining utility loss in Figure 11 (c). As we can see, the predicted and observed values match at most times except for two spikes at time 2 and time 58. Going into the log of the runtime performance, we found several high latency spikes (+60ms compared to the normal 10ms) on the migrated workload during migration. This is because the migration process will lock 256 KB blocks for consistency purposes; hence if the workload tries to access these blocks, it will be delayed until the lock is released. The performance models fail to capture this scenario and we observe a mismatch between the predicted and observed utility values. However, these are only transient behavior in the system and will not affect the overall quality of SMART's decision.

Test 2: Unexpected Case

Similar to the sanity check test (Figure 10 (a)), we intentionally create a load surge from time 10 to time 50. SMART invokes throttling immediately and waits for about 3 minutes (time 13) till the utility loss to invoke migration is lower than the loss due to throttling. The migration operation executed from time 13 to 17 and the system experienced no utility loss once it was done. Similar to the previous test, the temporarily lesser utility loss without any action (shown in Figure 12) is due to the ex-

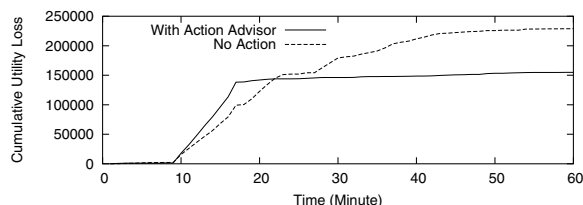


Figure 12: Spike Case: Cumulative Utility loss with migration invoked and without any action

tra utility loss for data movement. We skip other figures due to a lack of new observations for predicted and observed utility values.

4.4 Sensitivity Test

We test the sensitivity of SMART on the errors of performance prediction and future prediction in various configurations. This test is based on a simulator because it provides a more controlled and scalable environment, allowing us to test various system settings in a shorter time. We developed a simulator that takes the original system state, future workload forecasting, performance models and utility functions as input and simulates the execution of SMART's decisions. We vary the number of workloads in the system from 10 to 100. For each workload setting, 50 scenarios are automatically generated as follows:

Workload features: The sending rate and footprint size of each workload are generated using Gaussian mixture distribution: with a high probability, the sending rate (or footprint-size) is generated using a normal distribution with a lower mean value and a low probability, it is generated using another normal distribution with a larger mean value. The use of Gaussian mixture distribution is to mimic the real world behavior: a small number of applications contributes a majority of the system load and accesses the majority of data. Other workload characteristics are randomly generated.

Initial data placement: the number of components is proportional to the number of flows. For our tests, it is randomly chosen to be 1/10 of the number of flows. In addition, we intentionally create an un-balanced system (60% of the workloads will go to one component and the rest is distributed to other components randomly). This design is to introduce SLO violations and therefore, utility loss such that corrective actions are needed.

Workload trending: In addition, to mimic workload changes, 30% of workloads increases and 30% decreases. In particular, the daily growing step size is generated using a random distribution with mean of 1/10 of the original load and the decreasing step size is randomly distributed with mean of 1/20 of the original load.

Utility functions: The utility function of meeting the

SLO requirement for each workload is assumed to be a linear curve and the coefficients are randomly generated according to a uniform distribution ranging from 5 to 50. The utility function of violating the SLO requirement is assumed to be zero for all workloads. The SLO goals are also randomly generated with considerations of the workload demands and performance.

For a three month optimization window, the 500 scenarios experienced utility loss in various degrees ranging from 0.02% to 83% of the maximum system utility (the CDF is shown in Figure 13).

Test 1: With Accurate Performance Models and Future Prediction

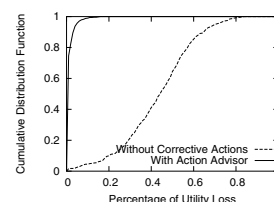


Figure 13: CDF of percentage of overall utility loss

For this test, we assume both the performance prediction and the future prediction are accurate. The Cumulative Distribution Functions of the *percentage of utility loss* (defined as $\frac{\text{utility_loss}}{\text{maximum_utility}}$) for both with and without corrective actions are shown in Figure 13. Comparing the two curves, with actions selected by Action Advisor, SMART is very close to the maximum utility. More than 80% scenarios have a utility loss ratio less than 2% and more than 93% have a ratio less than 4%.

Test 2: With Performance Model Errors

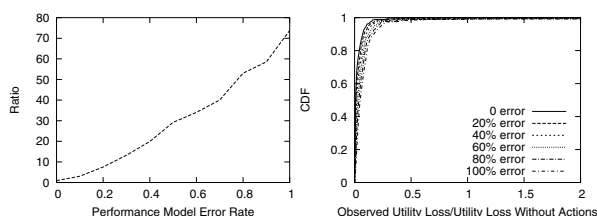


Figure 14: Impact of Model Errors (a) Observed_Utility_Loss / Predicted_Utility_loss (b) CDFs of Observed_Utility_Loss / Utility_Loss_Without_Action

Our previous analysis is based on the assumption that perfectly accurate component models are available for extrapolating latency for a given workload. However, this is not always true in real-world systems. To understand how performance prediction errors affect the quality of the decision, we perform the following experiments: (1) we generate a set of synthetic models and make decisions based on them. The latency calculated using these models is used as the “predicted latency”

and the corresponding utility loss is the “predicted utility loss”. (2) For the exact settings and action parameters, the “real latency” is simulated by adding errors on top of the “predicted latency” to mimic the real system latency variations. Because the residual normally grows with the real value, we generate a random scaling factor rather than the absolute error. For example, if the predicted latency is 10ms and the random error is 0.2, the real latency is simulated as $10 \times (1 + 0.2) = 12\text{ms}$. The “real utility” is estimated based on this.

Figure 14 (a) shows the ratio of $\frac{\text{observed_utility_loss}}{\text{predicted_utility_loss}}$, which reflects the degree of mismatch between the predicted and observed utility loss due to performance prediction errors. As we can see, there is significant difference between them – for a 20% model error, the average *observed_utility_loss* is 6 times of the predicted value. Next, we examine how does this difference affect the quality of the decision? Figure 14(b) plots the CDF of the remaining percentage of utility loss, defined as $\frac{\text{observed_utility_loss}}{\text{utility_loss_without_actions}}$. It grows as the model error increases. But even with a 100% model error, on average, the action selected by Action Advisor removes nearly 88% of the utility loss.

Test 3: With Time Series Prediction Errors

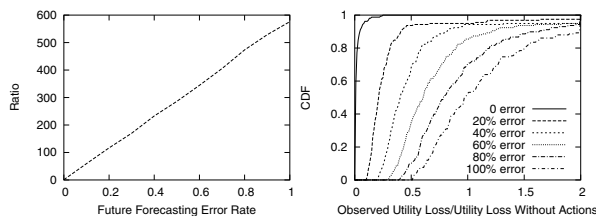


Figure 15: Impact of Future Forecasting Errors (a) *Observed_Utility_Loss* / *Predicted_Utility_Loss* (b) CDFs of *Observed_Utility_Loss* / *Utility_Loss_Without_Action*

The future forecasting error is introduced in a similar fashion: the workload demands forecasting is generated first. Based on that, the “real” workload demands is generated by scaling with a random factor following a normal distribution, with the future forecasting error as the standard deviation. The *predicted* utility loss is calculated based on the “forecasted” demands and the *observed* utility loss is calculated based on the “real” workload demands. Note that, for this set of tests, we restrict the Action Advisor only operating in the *normal mode* because otherwise, Action Advisor will automatically switch to the *unexpected mode*.

Figure 15 (a) shows the ratio of the *observed_utility_loss* to the *predicted_utility_loss*. It is much bigger than that of the model error: 560 vs 75 with 100% error. This is because the future forecasting error has a larger scale. For example, an error of under-estimating 10% of the future demands may lead to 100 IOPS under-estimated, which may

offset the performance prediction even more than the one caused directly by model errors. Figure 15 (b) shows the CDF of remaining utility loss after invoking Action Advisor’s action schedule. It shows that as the future forecasting error grows, the probability that the Action Advisor’s decision is helpful reduces very quickly. It can even result in a utility loss higher than doing nothing. This confirms our design choice: when the difference between the future prediction and observed values are high, we should apply a defensive strategy rather than operating in the normal mode.

Comparing the results of model and future forecasting errors, the quality of the decision is more sensitive to future forecasting accuracy than to model accuracy. In addition, we have used the ARIMA algorithm to perform time-series analysis on HP’s Cello99 real world trace and the results show that more than 60% of the predictions falls within 15% of the real value and more than 80% falls within 20%.

5 Related Work

Storage virtualization can be host based [7], network based [23] or storage controller based [28]. Storage virtualization can be at file level abstraction [13] or block level abstraction [14]. These storage virtualization solutions provide support for automatically extending volume size. However, these virtualization solutions do not offer multi-action based SLO enforcement mechanism and only recently single action based (workload throttling) SLO enforcement mechanisms are being combined with storage virtualization solutions [9]. SLO enforcement can also be performed by storage resource management software such as control center from EMC [1] and total productivity center from IBM [2]. These management software frameworks provide sensor and actuator frameworks, and they also have started to provide problem analysis and solution planning functionality. However, the current versions of these products do not have the ability to combine multiple SLO enforcement mechanisms.

Research prototypes that provide single action based solutions to handle SLO violations exist for a range of storage management actions. Chameleon [25], SLE-DRunner [9], and Facade [17] prototypes provide workload throttling based SLO enforcement solutions. Qos-Mig [10] and Aqueduct [16] provide support for data migration based SLO enforcement solutions. Ergastulum [6], Appia [27] and Minerva [3] are some capacity planning tools that can be used to either design a new infrastructure or extend an existing deployment in order to satisfy SLOs.

Hippodrome [5] is a feedback based storage management framework from HP that monitors system behavior and comes up with a strategy to migrate the system from

the current state to the desired state. Hippodrome focuses on migrating data and re-configuring the system to transform it from its current state to the new desired state.

6 Conclusion and Future Work

SMART generates a combination of corrective actions. Its action selection algorithm considers a variety of information including forecasted system state, action cost-benefit effect estimation and business constraints, and generates an action schedule that can reduce the system utility loss. It also can generate action plans for different optimization windows and react to both expected load surges and unexpected ones. SMART's prototype has been implemented in a file system. Our experiments show that the system utility value is improved as predicted. Experimental results show that SMART's action decision can result in less than 4% of utility loss. Finally, it is important to note that this framework can also be deployed as part of storage resource management software or storage virtualization software. Currently, we are designing a robust feedback mechanism to handle various uncertainties in production systems. We are also developing pruning techniques to reduce the decision making overhead, making SMART applicable in large data center and scientific deployments.

Acknowledgments

We want to thank John Palmer for his insights and comments on earlier versions of this paper. We also want to thank the anonymous reviewers for their valuable comments. Finally, we thank the HP Labs Storage Systems Department for making their traces available to the general public.

References

- [1] EMC ControlCenter family of storage resource management (SRM). <http://www.emc.com/products/storage-management/controlcenter.jsp>.
- [2] IBM TotalStorage. <http://www-1.ibm.com/servers/storage>.
- [3] ALVAREZ, G. A., BOROWSKY, E., GO, S., ROMER, T. H., BECKER-SZENDY, R., GOLDING, R., MERCHANT, A., SPA-SOJEVIC, M., VEITCH, A., AND WILKES, J. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems* 19, 4 (2001), 483–518.
- [4] ANDERSON, E. Simple table-based modeling of storage devices. Tech. Rep. HPL-SSP-2001-4, HP Laboratories, July 2001.
- [5] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: Running circles around storage administration. *Proceedings of Conference on File and Storage Technologies (FAST)* (Jan. 2002), 175–188.
- [6] ANDERSON, E., KALLAHALLA, M., SPENCE, S., SWAMINATHAN, R., AND WANG, Q. Ergastulum: an approach to solving the workload and device configuration problem. Tech. Rep. HPL-SSP-2001-5, HP Laboratories, July 2001.
- [7] ANONYMOUS. Features of veritas volume manager for unix and veritas file system. <http://www.veritas.com/us/products/volumemanager/whitepaper-02.html> (2005).
- [8] AZOFF, M. E. Neural network time series forecasting of financial markets.
- [9] CHAMBLISS, D., ALVAREZ, G. A., PANDEY, P., JADAV, D., XU, J., MENON, R., AND LEE, T. Performance virtualization for large-scale storage systems. *Proceedings of the 22nd Symposium on Reliable Distributed Systems* (Oct. 2003), 109–118.
- [10] DASGUPTA, K., GHOSAL, S., JAIN, R., SHARMA, U., AND VERMA, A. Qosmig: Adaptive rate-controlled migration of bulk data in storage systems. *ICDE* (2005).
- [11] DIEBOLD, F. X., SCHUERMANN, T., AND STROUGHAIR, J. Pitfalls and opportunities in the use of extreme value theory in risk management.
- [12] GANGER, G. R., WORTHINGTON, Y. N., AND PATT, B. L. A. The DiskSim simulation environment version 1.0 reference manual. Tech. Rep. CSE-TR-358-98, 27 1998.
- [13] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. Google file system. *Proceedings of SOSP* (2003).
- [14] GLIDER, J., FUENTE, F., AND SCALES, W. The software architecture of a san storage control system. *IBM System Journal* 42, 2 (2003), 232–249.
- [15] KARP, R. On-line algorithms versus off-line algorithms: how much is it worth to know the future? In *Proceedings of IFIP 12th World Computer Congress* (1992), vol. 1, pp. 416–429.
- [16] LU, C., ALVAREZ, G. A., AND WILKES, J. Aqueduct: on-line data migration with performance guarantees. *Proceedings of Conference on File and Storage Technologies (FAST)* (Jan. 2002), 175–188.
- [17] LUMB, C., MERCHANT, A., AND ALVAREZ, G. Facade: Virtual storage devices with performance guarantees. *Proceedings of 2nd Conference on File and Storage Technologies (FAST)* (Apr. 2003), 131–144.
- [18] RUEMLER, C., AND WILKES, J. A trace-driven analysis of disk working set sizes. Tech. Rep. HPL-OSR-93-23, Palo Alto, CA, USA, May 1993.
- [19] RUEMLER, C., AND WILKES, J. An introduction to disk drive modeling. *Computer* 27, 3 (1994), 17–28.
- [20] RUSSELL, S., AND NORVIG, P. *Artificial Intelligence A Modern Approach*. Prentice Hall, 2003.
- [21] SCHMUCK, F., AND HASKIN, R. Gpfs: A shared disk file system for large computing clusters, 2002.
- [22] SHRIVER, E., MERCHANT, A., AND WILKES, J. An analytic behavior model for disk drives with readahead caches and request reordering. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems* (New York, NY, USA, 1998), ACM Press, pp. 182–191.
- [23] TATE, J., BOGARD, N., AND JAHN, T. Implementing the ibm totalstorage san volume controller software on the cisco mds 9000. *IBM Redbook SG24-7059-00* (2004).
- [24] TRAN, N., AND REED, D. A. ARIMA time series modeling and forecasting for adaptive i/o prefetching. *Proceedings of the 15th international conference on Supercomputing* (2001), 473–485.
- [25] UTTAMCHANDANI, S., YIN, L., ALVAREZ, G., PALMER, J., AND AGHA, G. Chameleon: a self-evolving, fully-adaptive resource arbitrator for storage systems. *Proceeding of Usenix Annual Technical Conference (Usenix)* (June 2005).
- [26] WANG, M., AU, K., AILAMAKI, A., BROCKWELL, A., FALOUTSOS, C., AND GANGER, G. R. Storage device performance prediction with CART models. *SIGMETRICS Perform. Eval. Rev.* 32, 1 (2004), 412–413.
- [27] WARD, J., O'SULLIVAN, M., SHAHOURNIAN, T., WILKES, J., WU, R., AND BEYER, D. Appia and the hp san designer: automatic storage area network fabric design. In *HP Technical Conference* (Apr. 2003).
- [28] WARRICK, C., ALLUIS, O., AND ETAL. The ibm totalstorage ds8000 series: Concepts and architecture. *IBM Redbook SG24-6452-00* (2005).
- [29] WILKES, J. The pantheon storage-system simulator. Tech. Rep. HPL-SSP-95-14, HP Laboratories, dec 1995.

sMonitor: A Non-Intrusive Client-Perceived End-to-End Performance Monitor of Secured Internet Services

Jianbin Wei and Cheng-Zhong Xu

Dept. of Elec. and Comp. Engg., Wayne State University, Detroit, MI, 48202

Email: {jbwei, czxu}@wayne.edu

Abstract

End-to-end performance measurement is fundamental to building high-performance Internet services. While many Internet services often operate using HTTP over SSL/TLS, current monitors are limited to plaintext HTTP services. This paper presents sMonitor, a non-intrusive server-side end-to-end performance monitor that can monitor HTTPS services. The monitor passively collects live packet traces from a server site. It then uses a size-based analysis method on HTTP requests to infer characteristics of client accesses and measures client-perceived pageview response time in real time. We designed and implemented a prototype of sMonitor. Preliminary evaluations show measurement error of less than 5%.

1 Introduction

The Internet is increasingly being used as a platform to deliver information to remote clients in a secure and timely manner. End-to-end performance monitoring is fundamental to continuously supporting and developing such Internet services. For example, performance monitoring is critical for provisioning of quality of service guarantees. Without accurate measurement of client-perceived service quality, it will be impossible to allocate server resources between different clients.

To effectively support the developing of Internet services, there are three requirements in performance monitoring. First, a performance monitor needs to be non-intrusive so as to minimize its interference with operations of monitored systems. While Web server and Web page instrumentations, as proposed in [2, 11, 19], are easy to be deployed in small Web sites, they need to modify Web servers and Web pages. Thus they are not suitable for large-scale Internet services and are limited to certain Web servers. Second, the monitor should be able to characterize service performance perceived by all clients without biases. Active sampling of Internet services, such as that provided by Keynote [13], can obtain detailed response time characteristics from particular network locations. These characteristics, however, are not representative since they use customized browsers, which are different from that are used by real clients.

The browser-instrumentation approaches, such as Page Detailer from IBM [12], also have the same limitations as active sampling and are mainly for service testing and debugging. Therefore, a performance monitor should be close to servers to capture all traffic in and out the servers. Third, to accommodate the increasing deployment of HTTPS services, such as e-commerce service, the monitor should be able to measure performance of HTTPS services as well. Although recently proposed EtE [6] and ksniffer [17] can meet the first two requirements, they fail the last one since they need to parse the HTTP headers, which are unavailable for encrypted HTTPS traffic.

As a remedy, we present sMonitor, a non-intrusive server-side performance monitor that is capable of monitoring client-perceived end-to-end response time of HTTPS services in real time. sMonitor resides near monitored servers and passively collects traffic in and out of the servers. It then applies a size-based analysis method on HTTP requests to infer characteristics of client accesses. The size analysis is based on our observations that the first HTTP request to retrieve the base object, which normally is an HTML file, of a Web page is normally significantly larger than the following requests for the page's embedded objects. The size difference is because they have different `Accept` headers. Based on the inferred characteristics, sMonitor measures client-perceived pageview response time. Furthermore, sMonitor calculates the size of an HTTP request solely based on plaintext IP, TCP, and SSL/TLS [1, 9] packet headers. Thus there is no need to parse HTTP headers and sMonitor supports performance monitoring of HTTPS services.

This paper describes the design and implementation of sMonitor, and presents its preliminary evaluation. To conduct this evaluation, we used sMonitor to monitor service performance of HTTP and HTTPS Web servers. The evaluation results showed that the measurement error was less than 5%.

The structure of the paper is as follows. Section 2 discusses the size relationship between HTTP and HTTPS messages. Section 3 presents the design of sMonitor and

its implementation. Section 4 presents preliminary evaluations and Section 5 concludes the paper.

2 HTTP over SSL/TLS

In general, secured Internet services use either a version of Secure Sockets Layer (SSL) protocol [9] or Transport Layer Security (TLS) protocol [1] to encrypt HTTP messages before transmitting them over TCP connections. Since SSL and TLS are very similar, our discussion of SSL also applies to TLS unless specified explicitly.

In the SSL protocol, HTTP request messages are first fragmented into blocks of 2^{14} bytes or less. Next, compression is optionally applied. A message authentication code (MAC) over the compressed data is then calculated using HMAC MD5 or HMAC SHA-1 algorithms [14] and is appended to the fragment. After that, the compressed message plus the MAC are encrypted using symmetric encryption. The final step of the SSL record protocol is to attach a 5-byte plaintext header to the beginning of the encrypted fragment. The SSL record data are then passed to lower protocols, such as TCP, for transmission. In this way, with the support of SSL, all HTTP messages are transmitted with a guarantee of their secrecy, integrity, and authenticity.

As is known, it is extremely hard to hide information such as the size or the timing of messages [7]. We conducted experiments to determine the size relationship between HTTP and HTTPS messages. In the experiments, we sent HTTP requests with known sizes over SSL to an HTTPS Web server using Microsoft's Wininet library. The corresponding HTTPS messages were captured using WinPcap and their sizes were measured. There are three issues worth noting in the design of the experiments.

- In our experiments, to control the size of an HTTP request, we set the `Accept` header to meaningless characters. In HTTP/1.1 protocol [8], it specifies that a Web server should send a 406 ("not acceptable") response if the acceptable content type set in a request's `Accept` header cannot be satisfied directly. We found that, however, most Web servers, including Microsoft Internet Information Services (IIS) and Apache Web server, just ignore the unrecognized `Accept` header and send the default response. Thus the meaningless `Accept` header we set in an HTTP request does not bias our experimental results.
- In order to evaluate different protocols, encryption algorithms, and MAC algorithms, we modified the settings of Windows according to [15]. In practice, in IE 6.0 TLS is disabled while in IE 7.0 Beta 1 it is enabled by default. In addition, RC4 and MD5 are always used as the first option for encryption algorithm and MAC algorithm, respectively.

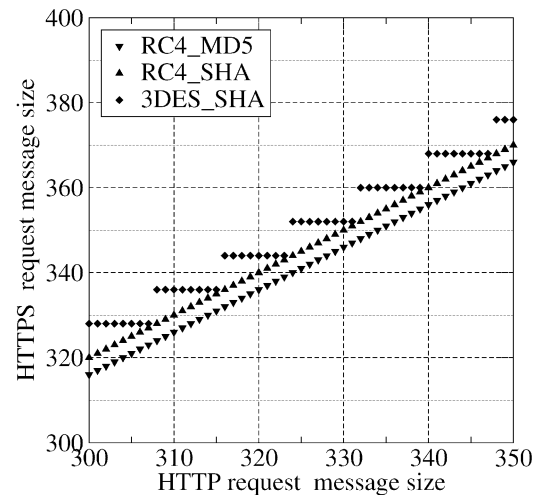


Figure 1: Size relationship between HTTP and HTTPS messages.

- To ensure the HTTPS requests are sent to the server instead of the local cache, we passed flags `INTERNET_FLAG_NO_CACHE_WRITE` and `INTERNET_FLAG_RELOAD` to function `HttpOpenRequest()` to disable caching.

We conducted experiments in which the size of an HTTP request ranges from 100 through 2000 bytes. The size of corresponding HTTPS message is obtained by checking the record header of the first SSL/TLS record fragment with content type as application data. To clearly show the size relationship between HTTP and HTTPS messages, Fig. 1 only depicts the results where the HTTP message size ranges from 300 through 350 bytes. In this figure, RC4_MD5 denotes that RC4 encryption algorithm and MD5 MAC algorithm are used. Legend RC4_SHA and 3DES_SHA follow similar format. We obtained the cipher suite used between the client and the server by checking their handshake messages for session establishment, which are not encrypted.

From Fig. 1 we observe that the size difference between HTTPS and HTTP request messages is always 16 bytes and 20 bytes for RC4_MD5 and RC4_SHA, respectively. It is because RC4 is a stream cipher, in which the ciphertext has the same size as plaintext, and MD5 and SHA calculate a 16-byte and 20-byte MAC, respectively.

In the case of 3DES_SHA, we can infer the size of an HTTP message within 8 bytes from the size of the corresponding HTTPS message. Assuming a 360-byte HTTPS message, since there exists a padding size byte, only another zero or seven bytes need to be padded to make the total an integer multiple of 8 in 3DES. Considering the 20-byte MAC calculated by SHA, the HTTP message therefore ranges from 332 through 339 bytes. It can be observed from Fig. 1.

In SSL 3.0, the padding added prior to encryption of

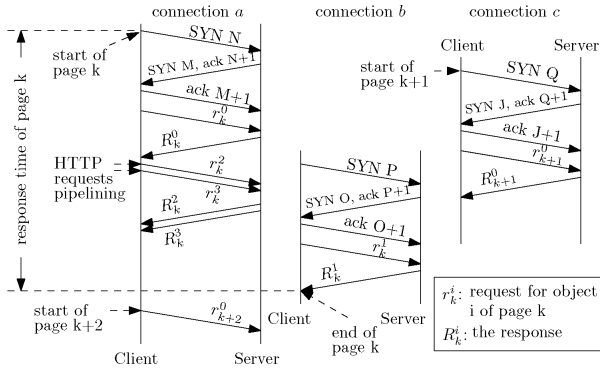


Figure 2: Retrievals of multiple pages and embedded objects using pipelined requests over multiple connections.

an HTTP message is the *minimum* amount required so that the total size of the data to be encrypted is a multiple of the cipher's block size. In contrast, TLS 1.0 and 1.1 (draft) define *random* padding in which the padding can be any amount that results in a total that is a multiple of the cipher's block size, up to a maximum of 255 bytes. It is to frustrate attacks based on size analysis of exchanged messages. From Fig. 1 we observe that random padding is not implemented in IE 6.0 and 7.0 Beta 1. We also verified this observation for TLS used in Firefox.

If compression is performed, the size of HTTP requests will change, which might interfere the size-based analysis method presented in Section 3. However, from Fig. 1 we observe that the optional compression step in SSL/TLS is not performed. It is because no default compression algorithm is defined in SSL and TLS protocols.

In summary, we can obtain the size of an HTTP request from the corresponding HTTPS message in case of stream ciphers, such as RC4, or an estimation within 8 bytes difference in case of block ciphers, such as DES and 3DES. This agrees with the observations made in [21]. Therefore, we can conduct size analysis on HTTPS traffic to measure the response time of HTTPS services.

3 Design and Implementation of sMonitor

In general, a Web page consists of a base object and multiple embedded objects. A browser retrieves a Web page by issuing a series of HTTP requests for all objects. The key challenge in end-to-end performance monitoring is to determine the beginning and the end of the retrieval of a Web page. The determination needs to be achieved even when multiple Web pages are retrieved by clients using pipelined HTTP requests over multiple TCP connections. The scenario is depicted in Fig. 2. More important, in HTTPS services, such determination must be achieved without parsing HTTP headers, which are encrypted.

Let r_k^0 and r_k^i denote the HTTP requests for the base

object and embedded object i of some Web page k , respectively. Essential to the inference of client-perceived response time is the identification of request r_k^0 , which delimits Web page retrievals. In this section we present the size-based analysis method to identify r_k^0 . As discussed in Section 2, the size-based analysis method also works for HTTPS services.

3.1 Analysis of HTTP Accept header

Our size-based analysis method is based on the HTTP `Accept` header, which is designed as part of the effort for content negotiation. However, HTTP/1.0 and HTTP/1.1 do not specify how browsers should implement the `Accept` header. In this work, we focus on two most popular Web browsers, Microsoft Internet Explorer (IE 6.0 and 7.0 Beta 1) and Mozilla Firefox 1.0 and 1.5 in Windows, which have more than 95% market share [16]. Notice that there is no need for the sMonitor to know the browsers used by clients in measuring HTTPS service performance.

In IE, the default `Accept` header is specified in the registry key `Accepted Documents` as part of local machine's Internet settings with size of 56 bytes. When certain applications are installed, their identifications may be added to the registry key. In Firefox, the `Accept` header can be accessed through URL `about:config` and its default value is 99 bytes. There are also several commonly used `Accept` headers in Firefox. Their sizes range from 3 through 56 bytes.

Based on our experiments, we have following important observations regarding the use of `Accept` header in HTTP requests.

- In an HTTP request, the default `Accept` header is normally used in IE if the requested object is to be loaded into a known frame or in Firefox if the object is to be loaded into any frame as the first object, regardless of the content type of the object. Otherwise, in IE the short header “*/*” is normally used or in Firefox the `Accept` header is set according to the object's content type.

A known frame means it is specified using reserved HTML target names “_self”, “_parent”, or “_top”; or the target name can be found in the window that contains the link or every other window. Otherwise, the frame is unknown. A frame created using target name “_blank” is always an unknown one.

3.2 Size analysis for request identification

Based on the observations, we found that request r_k^0 normally was significantly larger than request r_k^i while the difference between r_k^i and r_k^{i+1} is small. An HTTP request message begins with a request line and is followed by a set of optional headers and optional message body.

The large size difference between r_k^0 and r_k^i is mainly because the following reasons.

- r_k^0 has larger `Accept` header than r_k^i . For example, in IE, the size difference of `Accept` header between default one and the short one is at least 53 bytes and can become even larger than 161 bytes if more software is installed in the client system. In Firefox, the size difference is at least 43 bytes and can be as large as 93 bytes.
- For static Web objects, the request lines in r_k^0 and r_k^i normally have limited size for the ease of Web site administration. For dynamically generated objects, their addresses normally follow the same pattern and the sizes are very close to each other.
- The only difference between requests r_k^0 and r_k^i normally is the `Accept` headers if cookie is not used. In the case when cookie is used, a `Cookie` header will be used in all requests except the first one from a client. Since normally the first request is for base object of a Web page, we also identify it as r_k^0 . Thus, the size difference is a good indicator for different requests.
- Across a wide variety of measurement studies, the overwhelming majority of Web requests use the GET method to retrieve Web pages and invoke scripts [10, 18]. They do not have message bodies.

Based on the observation of size difference, we propose a size-based analysis method to identify request r_k^0 . Let x_n denote the size of the n th HTTP request message. Let $f(x) = x$ denote the HTTP request size function. Its second derivative can be approximated as

$$\begin{aligned} f''(x_n) &\cong (f'(x_n + h/2) - f'(x_n - h/2))/h \\ &= (f(x_{n+1}) - 2f(x_n) + f(x_{n-1}))/h^2 + O(h^2). \end{aligned}$$

Since the function is discrete, let h be 1, we have

$$f''(x_n) \cong x_{n+1} - 2x_n + x_{n-1}.$$

If $f''(x_n) < 0$, then $f(x)$ is concave down (\cap) and it has a relative maximum at x_n . Let t denote a configurable threshold of the size difference between r_k^0 and r_k^i . We have

- request n is identified as r_k^0 if its second derivative $f''(x_n)$ is less than t .

The selection of t should maximize the number of correctly detected r_k^0 and limit the number of false positive ones that would otherwise damage the accuracy of sMonitor. We found that the `Accept` headers issued by Firefox have smaller size differences than those in IE. Based on analysis of `Accept` header and captured HTTP requests, we set t as -60 bytes in sMonitor. We find that it is a good setting through our experiments.

To measure client-perceived response time, we need to consider following issues. First, we must consider the

time difference between sending segments from clients and receiving them in servers and vice versa. Normally, when a client issues request r_k^0 and there is no connection between the client and the server, the client must first contact the server for connection establishment using an SYN segment. Request r_k^0 is normally issued right after the last step of the three-way handshake. Therefore, there is at least 1.5 round-trip time (RTT) gap before the server receives the first r_k^0 segment. In the case that request r_k^0 is transmitted over an established TCP connection, the time gap then is 0.5RTT. There is also a 0.5RTT time gap between the server sending the last packet of a response and the client receiving the packet.

Second, in sMonitor, the end of a Web page is identified as the last non-empty server segment before another request r_k^0 over the same or other connections between the client and the server. This identification method may cause delayed identification because there exists a time gap between the end of a Web page and the arrival of requests for next Web page. We address this issue using a configurable time-out scheme. If there is no any new segment from either the client or the server, sMonitor determines that the Web page is ended. In sMonitor, we adopt 5 seconds by default as it is treated as good response time of a Web page [4].

In [5], the authors marked the first packet with size smaller than the MTU as the end of server response. They argued that a server response is packed into a series of packets with size as the MTU except the last one because there is not enough content to fill it. The method, however, is problematic because the actual size of a packet is affected by multiple factors, including the TCP implementation in the server and network conditions between the client and the server. It becomes even worse in the case of packet retransmission and reordering. For example, assuming a server response is packed into an MTU-size packet and a non-MTU size packet, and they are sent to the client together without receiving any acknowledgments. The first packet can be lost and retransmitted after the second one. Their method then determines false end of the server response.

Third, in sMonitor we also need to consider the effects of HTTP pipelining. In HTTP pipelining, multiple requests can be issued to the server before the response of the first request is sent out. We found that HTTP/1.1 pipelining was becoming widely used. We examined three-hour traffic in and out the Web server of the college of engineering at wayne state university. The results showed that 19.3% of HTTP requests were pipelined. Also, pipelining is implemented in Firefox as an experimental feature. As observed in [20], more than 90% requests are less than 1000 bytes and most large requests have non-GET methods for services such as web-mail. Thus, if the segment size is the same as the

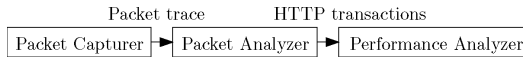


Figure 3: The architecture of sMonitor.

MTU, sMonitor assumes that this is the first segment of a request and more are coming. Otherwise, the monitor marks the segment as the end of a request.

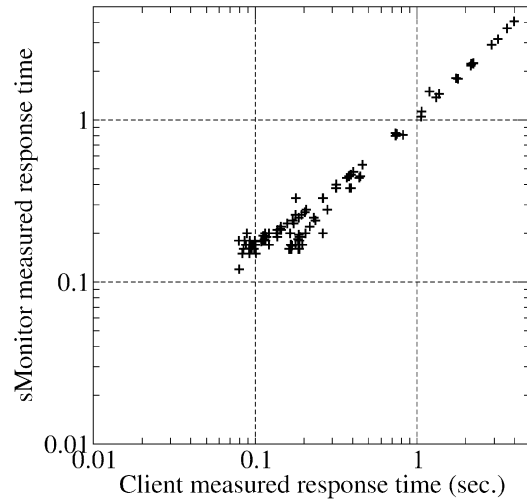
Fourth, we also need to deal with parallel downloading, which means a user retrieves multiple Web pages from one server at the same time. During parallel downloading, the requests for these pages can intertwine each other, which might cause false identification of the end of a Web page. As an example, assuming that the user accesses Web pages k and $k + 1$ simultaneously. In sMonitor, all requests issued after request r_{k+1}^0 are identified as the requests for embedded objects in page $k + 1$ while request r_k^i may come after r_{k+1}^0 . Consequently, the measured response time of page k is smaller than that perceived by the user while it is opposite for page $k + 1$. However, sMonitor still measures average response time perceived by the client accurately. The parallel downloading of embedded objects within a Web page does not affect the performance of sMonitor because it does not affect the identification of the beginning of r_k^0 for the Web page.

3.3 Implementation of sMonitor

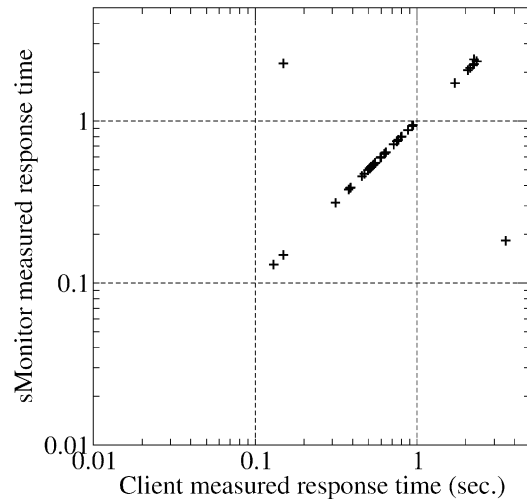
A prototype of sMonitor has been implemented at user-level as a stand-alone application in C. It captures packets in and out of the monitored servers, analyzes packet headers to extract packet information, derives page-related data from packet headers, and conducts performance analysis. Fig. 3 presents the architecture of sMonitor. The *packet capturer* collects live network packets using pcap [22] (libpcap on Unix-like systems or WinPcap on Windows). In the *packet analyzer*, sMonitor parses the packet headers to extract HTTP transaction information, such as HTTP request sizes, and passes them to the *performance analyzer*. Such information can be obtained by parsing TCP/IP and SSL/TLS headers. The *performance analyzer* derives client-perceived response time of the monitored services.

4 Preliminary Evaluation

To evaluate the accuracy of sMonitor, we conducted two experiments and compared the response time measured by sMonitor with those measured on client side. In the first experiment, we modified Surge [3] so that the first requests for Web pages have long Accept headers. We also modified Surge so that it records the pageview response time. We then set up a Web server in Detroit, MI. Surge was deployed in a node of PlanetLab in UCSD to access the Web server. In the experiment, around 140 Web pages were retrieved from the server in 10 minutes.



(a) PlanetLab evaluation.



(b) Evaluation with real services.

Figure 4: Accuracy evaluation of sMonitor.

sMonitor was deployed on the Web server to capture the network traffic and measure the response time.

Fig. 4(a) shows the comparison between the client-side measurement and the sMonitor measurement. From this figure we can clearly see that these two measurements have strong linear relationship since the sMonitor measured response times are very close to those recorded by Surge directly. On the other hand, when the response time is smaller than 1 second, we can see that the measurement difference is relatively large. This is mainly caused by the variance in the estimated RTT. A numerical analysis shows that the average response times measured by Surge and sMonitor are 1.00 and 0.95 seconds, respectively. The difference is 5%.

In the second experiment, we used IE to access 44 Web pages in several real Web services, including several banks' Web sites. We also recorded the correspond-

ing response time manually. The sMonitor was placed on the client to measure the response time.

The results are plotted in Fig. 4(b). Comparing with Fig. 4(a) we can see that the measured response times are very accurate even when the response times are smaller than 1 second. This is because the client-side placement of sMonitor removes the variance in RTT estimations. From this figure we can also see several falsely identified Web pages. For example, when the client-perceived response time is 3.5 seconds, sMonitor measures the response time as 0.18 seconds since it incorrectly delimits the beginning and ending of the Web page. Similarly, when the client-perceived response time is 0.15 seconds, sMonitor gives the response time as 2.27 seconds. However, from the figure we can see that in most cases the Web pages are correctly delimited. The average response times recorded manually and those by sMonitor are 1.02 and 0.99 seconds, respectively. The difference is 3%. Therefore, from these two experiments we conclude that sMonitor is very accurate in measuring the client-perceived pageview response time.

5 Conclusions

In this paper we presented sMonitor to monitor end-to-end performance of HTTPS services. We designed a size-based analysis method on HTTP requests to characterize the client access behaviors. Based on the characteristics, we designed and implemented a prototype of sMonitor. We investigated its accuracy in measuring client-perceived response time using HTTPS and HTTP services and found that the difference was less than 5%.

We note that the accuracy evaluation is preliminary. The environments of real Web services are very complicated. In the future, we plan to conduct comprehensive evaluation of our approach using live Web services. The size-based analysis method may also be improved. For example, we used simple heuristic methods to address the issues of parallel downloading and HTTP pipelining. In the future, we will work on systematic solutions for these issues.

Acknowledgements

We would like to thank our shepherd, Erich Nahum, for helping us to significantly improve this work and the anonymous reviewers for their numerous helpful comments. This work was supported in part by US NSF grants ACI-0203592, CCF-0611750, and NASA 03-OBPR-01-0049.

References

- [1] ALLEN, C., AND DIERKS, T. *The TLS Protocol Version 1.0*. RFC 2246, January 1999.
- [2] ALMEIDA, J., DABU, M., MANIKUTTY, A., AND CAO, P. Providing differentiated levels of service in web content hosting. In *Proc. ACM SIGMETRICS Workshop on Internet Server Performance* (June 1998), pp. 91–102.
- [3] BARFORD, P., AND CROVELLA, M. Generating representative Web workloads for network and server performance evaluation. In *Proc. ACM SIGMETRICS conference* (June 1998), pp. 151–160.
- [4] BHATTI, N., BOUCH, A., AND KUCHINSKY, A. Integrating user-perceived quality into Web server design. In *Proc. Int'l World Wide Web Conference* (2000), pp. 1–16.
- [5] CHENG, H., AND AVNUR, R. Traffic analysis of SSL encrypted Web browsing. Available at: <http://www.cs.berkeley.edu/~daw/teaching/cs261-f98/projects/final-reports/ronathan-heyning.ps>.
- [6] CHERKASOVA, L., FU, Y., TANG, W., AND VAHDAT, A. Measuring and characterizing end-to-end internet service performance. *ACM Trans. on Internet Technology*, 3(4) (2003), 347–391.
- [7] FERGUSON, N., AND SCHNEIER, B. *Practical Cryptography*. John Wiley & Sons, 2003. p114.
- [8] FIELDING, R. T., GETTYS, J., MOGUL, J. C., NIELSEN, H. F., MASINTER, L., LEACH, P. J., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.
- [9] FREIER, A. O., KARLTON, P., AND KOCHER, P. C. The SSL protocol version 3.0, November 1996. <http://wp.netscape.com/eng/ssl3/ssl-toc.html>.
- [10] HERNANDEZ-CAMPOS, F., JEFFAY, K., AND SMITH, F. D. Tracking the evolution of web traffic: 1995-2003. In *Proc. Int'l Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2003), pp. 16–25.
- [11] HEWLETT-PACKARD DEVELOPMENT COMPANY, L.P. Openview transaction analyzer. <http://openview.hp.com/>.
- [12] IBM CORP.. Page detailer. <http://www.alphaworks.ibm.com/tech/pagedetailer>.
- [13] KEYNOTE SYSTEMS, INC. www.keynote.com.
- [14] KRAWCZYK, H., BELLARE, M., AND CANETTI, R. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104, February 1997.
- [15] MICROSOFT CORPORATION. How to restrict the use of certain cryptographic algorithms and protocols in schannel.dll, December 2004. <http://support.microsoft.com/?kbid=245030>.
- [16] NETAPPLICATIONS.COM. Firefox continues to erode microsoft dominance, June 2005. <http://www.netapplications.com/news.asp>.
- [17] OLSHEFSKI, D. P., NIEH, J., AND NAHUM, E. ksniiffer: Determining the remote client perceived response time from live packet streams. In *Proc. USENIX Operating Systems Design and Implementation* (2004).
- [18] PADMANABHAN, V. N., AND QIU, L. The content and access dynamics of a busy Web site: Findings and implications. In *Proc. ACM SIGCOMM* (2000), pp. 111–123.
- [19] RAJAMONY, R., AND ELNOZAHY, M. Measuring client-perceived response time on the WWW. In *Proc. USENIX Symp. on Internet Technologies and Systems* (March 2001).
- [20] SMITH, F. D., HERNANDEZ-CAMPOS, F., JEFFAY, K., AND OTT, D. What TCP/IP protocol headers can tell us about the Web. In *Proc. ACM SIGMETRICS* (2001), pp. 245–256.
- [21] SUN, Q., SIMON, D. R., WANG, Y.-M., RUSSELL, W., PADMANABHAN, V. N., AND QIU, L. Statistical identification of encrypted Web browsing traffic. In *Proc. IEEE Symp. on Security and Privacy* (May 2002).
- [22] TCPCDUMP. <http://www.tcpdump.org/>.

Privacy Analysis for Data Sharing in *nix Systems

Aameek Singh

Ling Liu

Mustaque Ahamad

College of Computing, Georgia Institute of Technology
{aameek, lingliu, mustaq}@cc.gatech.edu

Abstract

Linux and its various flavors (together called *nix) are growing in mainstream popularity and many enterprise infrastructures now are based on *nix platforms. An important component of these systems is the ingrained multi-user support that lets users share data with each other. In this paper, we analyze *nix systems and identify an urgent need for better privacy support in their data sharing mechanisms. In one of our studies it was possible to access over 84 GB of private data at one organization of 836 users, including over 300,000 emails and 579 passwords to financial and other private services websites. The most surprising aspect was the extremely low level of sophistication of the attack. The attack uses no technical vulnerabilities, rather inadequacies of *nix access control combined with user/application's privacy-indifferent behavior.

1 Introduction

With increasing popularity of open source operating systems like Linux, many enterprises are opting to set up their intranets using such *nix systems. The market research firm IDC expects linux server sales to hit US \$9.1 billion by 2008 and that linux servers will comprise 25.7% of total server units shipped in 2008. Consequently, many big companies like IBM, Dell actively support the open source *nix movement.

One of the important features of *nix systems is their ingrained multi-user support. These operating systems are designed for simultaneous multiple users and provide seamless mechanisms to share data between different users. For example, user *alice* can set up appropriate access "permissions" on the data she wants to share with her group *students* by executing a simple `chmod` command [7]. These access permissions are dictated by the *nix *access control model*.

In this paper, we take a critical look at the *nix access control model. Our objective is to analyze the **privacy**

support in its data sharing mechanisms. For example, how does the system assist a user to share data only with desired users and prevent private information from being leaked to unauthorized users? In order to successfully do this, we also need to look at the **convenience** of using *nix data sharing mechanisms in typical situations. This is due to the fact that lack of convenience typically leads to users compromising (intentionally or mistakenly) their security requirements to conveniently fit the specifications of the underlying access control model.

Please note that we use the seemingly oxymoronic phrase "*private sharing*" to indicate the desire of sharing data only with a select set of authorized users.

As part of our analysis, we looked at how users use the access control for their data sharing needs in practice. We conducted experiments at two *nix installations of a few hundred computer-literate users each. Surprisingly, we found that large chunks of private data was accessible to unauthorized users. In many cases, the user's definition of an "authorized user" does not match the underlying system's definition, that leads to such a breach. This observation is best exemplified in the following scenario. Many users attempt to privately share data by using execute-only permissions for their home directories. This prevents other users from listing the contents of the directory, but any user who knows the name of a subdirectory can `cd` into it. The data owner *authorizes* some users by explicitly giving them the names of the subdirectory through out-of-band mechanisms like personal communication or email. However, this authorization is not the same as the system's authorization. From the underlying system perspective, it is assumed that any user who issues the command with the right directory name is authorized. Thus, users who simply guess the subdirectory name can also access the data. Along with that, setting execute-only permissions on the home directory to share *one* subdirectory, puts all other subdirectories (sibling to the shared directory) also at risk, which if not protected appropriately, can be accessed.

We were able to effectively exploit these shortcomings in our studies. At one organization of 836 users, over 84 GB of data was accessible, including more than 300,000 emails and 579 passwords to financial websites like *bankofamerica.com* and other private websites like medical insurance records. Importantly, the attack does not always need to *guess* directory names, but can find actual names from unprotected command history files (*.history*, *.bash_history*) or standard application directory names (*.mozilla*). The reason for this surprisingly large privacy breach without exploiting technical vulnerabilities like buffer overflows or gaining elevated privileges, is the combination of lack of system support and user or even applications' privacy-indifferent behavior either mistakenly or for lack of anything better. Also, as we discuss later in Section-2, even for an extremely privacy-conscious user with all available tools, it is tough to protect private data in many situations.

The attack described in this paper is a form of an **insider** attack, in which the attacker is inside the organization. The attacker could be a disgruntled employee, contractor or simply a curious employee trying to access the salaries chart in the boss's home directory. According to a recent study by the US Secret Service and CERT [6], such attacks are on a rise with 29% of the surveyed companies reporting¹ having experienced an insider attack in the past year [4]. Also, in complete congruence to our attack, the report finds that:

"Most incidents were not technically sophisticated or complex ..."

The rest of the paper is organized as follows. In Section-2, we discuss multiple *nix issues, including usability, that lead to privacy breaches. In Section-3, we present our case studies at two *nix installations which demonstrate these breaches. We briefly describe possible enhancements to these breaches in Section-4. We conclude in Section-5.

2 Data Privacy: Vulnerability Analysis

In this section, we discuss various privacy breaches that occur in *nix systems. The discussion below primarily explores the popular {*owner*, *group*, *others*} *nix paradigm. The advanced mechanisms like ACLs [5] enhance privacy in only a few cases and we will demonstrate a clear need of a new, more complete solution.

2.1 Selective Data Sharing

The first privacy breach occurs due to the need to "*selectively*" share data. The selectivity can be of two kinds:

- **Data Selectivity:** Data selectivity is when a user wants to share only a few (say one) of the subdirectories in the home directory. So, an authorized user is allowed to access only the shared subdirectory, but not any of the sibling directories. In order to do this correctly, the owner needs to follow two steps - (a) set appropriate permissions to the shared subdirectory (at least execute permissions on the entire path to the subdirectory and the sharing permissions on the subdirectory), and (b) *remove permissions from the sibling subdirectories*. The second step is unintuitive, since the user needs to act on objects that are not the focus of the transaction. Also, any new file being created needs to be protected.
- **User Selectivity:** In many situations, users need to share data with an adhoc set of users that do not belong to a single user group or are only a subset of a user group, and not the entire *others* set. In this situation, the permissions for *group* or *others* are not sufficient. Creating a new group requires administrative assistance which is not always feasible.

In order to do selective data sharing (a common example is to allow access to `public_html`), currently owners mostly use execute-only permissions on the home directories. The perception is that since users can not list the contents of the directory, they cannot go any further than traversing into the home directory unless they know the exact name of the subdirectory. Now, the owner can authorize desired users by giving them the names of the appropriate subdirectories. Those *authorized* users can traverse into the home directory and then use the subdirectory name to `cd` into it (without having to list the contents of the home directory). From data selectivity perspective, it is assumed that they cannot access the rest of the contents and from user selectivity perspective, unauthorized users cannot access any contents.

However, the underlying system cannot distinguish between such authorized or unauthorized users. Any user who can **guess** the subdirectory name can actually access the data. For an attacker inside the organization, this is not a herculean task. For example, for a computer science graduate school, it is highly likely that users will have directories named *research*, *classes* or *thesis*. An easy way of creating such a list of names is by collecting names from users that actually have *read* permissions on the home directories. Within the context of a single organization, or in general human psychology, it is likely that many users have similar directory names.

Secondly, many times directory names do not need to be guessed at all. The names can be extracted from **history** files (like *.history* or *.bash_history*), that contain the commands last executed by the owner, like `cd`, which will include real directory names. In fact, in our experiments we found around 20-30% of all users had readable

history files and around 40% of the total leaked data was obtained from the directory names extracted from these history files.

Thirdly, it is not always user created directories that leak information. Many **applications** use standard directory names and fail to protect critical information. For example, the famous Mozilla web browser [2] stores the profile directory in `~/.mozilla` and had that directory world-readable [8] in many cases, till as late as 2003. Many *nix installations with the browser installed before that have this vulnerability and we were able to obtain around 575 password to financial and private websites (because users saved passwords without encrypting them). In addition, their browser caches, bookmarks, cookies and histories were also available. The browser Opera [3] also has a similar vulnerability, though to a lesser extent. While it can be argued that it is the responsibility of application developers to ensure that this does not happen, we believe that the underlying system can assist users and applications in a more proactive manner.

The POSIX ACLs [5], if used help in achieving only user selectivity. They do not address the data selectivity requirements or prevent leaking of application data.

2.2 Metadata Privacy

So far, we have only talked about the privacy breach for file *data*. However, there are many situations in which users are interested in protecting even the metadata of the files. The metadata contains information like ownership, access time, updation time, creation time and file sizes. There are scenarios where a user might obtain confidential information by just looking at the metadata. For example, an employee might be interested in knowing how big is his annual review letter or did the boss update it after the argument he had with her?

The *nix access control does not provide good metadata privacy. Even if users only have execute permissions on a directory, as long as they can guess the name of the contained file, its metadata can be accessed even if the file itself does not have any read, write or execute permissions on it. Thus, if a user has to share even a single file/directory within the home directory (thus, requiring atleast execute permissions), all other files contained in the home directory lose their metadata privacy if their names are known or can be guessed.

2.3 Data Sharing Convenience

User convenience is an important feature of an access control implementation. If users find it tough to implement their security requirements, they are likely to compromise them to easily fit the underlying access control model. This can be seen as one of the reasons why

encryption file systems are not in widespread use, even though they guarantee maximum security.

From our analysis of the *nix access control, along with some of the issues discussed earlier, we found the following two data sharing scenarios in which there is no convenient support for privacy.

- **Sharing a Deep-Rooted Directory:** For a user to share a directory that is multiple levels in depth from the home directory, there needs to be at least execute permissions on all directories in the path. This in itself (a) leaks the path information, (b) puts sibling directories at risk and (c) leaks metadata information for sibling directories. In order to prevent this, since most operating systems do not allow hard links to directories anymore, a user would have to create a new copy of the data. And since users are more careless with permissions for deep rooted directories (they protect a higher level directory and that automatically protects children directories), a copy of such a directory could have privacy-compromising permissions.
- **Representation of Shared Data:** In many circumstances the way one user represents data might not be the most suitable way for another user. For example, while an employee might keep his resume in a directory named `job-search`, it is clearly not the most apt name to share with his boss. The employee might want her to see the directory simply as CV. Changing the name to meet the needs of other users is not an ideal solution. This again shows the lack of adequate system support for private and convenient data sharing.

It is important to recognize that even an extremely privacy-conscious user can not protect data at all times. Exhaustive efforts to maintain correct permissions for all user and **application created** data will still be insufficient to protect metadata or allow private sharing of deep rooted directories with user-specific representation.

3 Case Studies

As part of our study, we conducted experiments at two geographically and organizationally distinct *nix installations. Users at both installations (CS graduate schools) are highly computer literate and can be expected to be familiar with all available access control tools. For our analysis, we consider the following data to be private:

- All user emails are considered private.
- All data under an execute-only home directory is considered private.
- Browser profile data (saved passwords, caches, browsing history, cookies) is considered private.

The second assumption above merits further justification. It can be argued that not every subdirectory under an execute-only home directory is meant to be private (for example, a directory named *public*). However, the semantics of the execute-only permission set dictate that any user other than the owner cannot list the contents of the directory and since the owner never *broadcasts* the names of the shared directories, an unauthorized user *should not* be able to access that data. And since we do not include in our measurements any obviously-private data from home directories of users with *read* permissions (for example, world-readable directories named *personal*, or *private*), we believe the two effects to approximately cancel out.

3.1 Modus Operandi

Next, we describe the design of our attack² that scans user directories and measures the amount of private data accessible to unauthorized users.

The attack works in multiple phases. The first step is to obtain directory name lists which can be tried against users with execute-only home directories. Three strategies are used to obtain these lists:

- **Static Lists:** These are manually entered names of directories likely to be found in the context of the organizations - CS graduate schools. For example, “*research*”, “*papers*”, “*private*” and their variants in case (“*Research*”) or abbreviations (“*pvt*”).
- **Global Lists:** These lists are generated by obtaining the directory names from home directories of users that have *read* permissions.
- **History Lists:** These are user specific lists generated by parsing users’ history files, if readable. We used a simple mechanism, parsing only `cd` commands with directory names. It is possible to do more by parsing text editor commands (like `vim`) or `copy/move` commands.

In the next step the tool starts a multi-threaded scanning operation that attempts to scan each user directory. For users with **no** permissions, no scanning is possible. For users with **read** permissions, as discussed earlier, since there is no precise way of guessing which data would be private, we only measure email and browser profile statistics. Finally, for users with **execute-only** permissions, along with email and browser statistics, we also attempt to extract data using the directory name lists prepared in the first step.

Evaluating Email Statistics

This is done by attempting to read data from standard mailbox names - “*mail*”, “*Mail*”, “*mbox*” in the home directory and the mail inboxes in `/var/mail/userName`.

A `grep` [7] like tool is used to measure (a) number of readable emails, (b) number of times the word “*password*” or its variants appeared in the emails.

Evaluating Execute-only Data Statistics

For users with execute-only permissions on the home directory, the scanner uses the combination of static, global and the user’s history lists to access possible subdirectories. Double counting is avoided by ensuring that a name appearing in more than one list is accounted for only once and by not traversing any symbolic links. While scanning the files, counts are obtained for the total number of files and the total size of the data that could be accessed.

Evaluating Browser Statistics

The mozilla browser [2] stores user profiles in the `~/.mozilla` directory. This directory used to be world-readable till as late as 2003 when the bug was corrected [8]. Within that profile directory, there are subdirectories for each profile that has been used by that user. Within the profile directory, there is another directory with a randomized name (for security against remote attacks) ending in “*.slt*” [1]. Within this directory, the following files exist and (with this bug) were readable:

- **Password Database:** A file with the name of type “*12345678.s*” containing user logins and passwords saved by mozilla. Since many users do not encrypt their passwords, mozilla stores the passwords in a base-64 encoding (indicated by the line starting with a `~` in the passwords file), which can potentially be trivially decoded to get plaintext passwords.
- **Cookies:** The `cookies.txt` file contains all browser cookies. Many websites including popular email services like Gmail, Hotmail allow users to automatically login by keeping such cookies. Hijacking this cookie can allow a malicious user to login into these accounts. For many other cookies related attacks, see [9].
- **Cache:** This is a subdirectory that contains the cached web pages visited by the user.
- **History Database:** Web surfing history, which many sophisticated viruses and spyware invest resources to collect, are also readable.

3.2 Results

The complete characteristics of the two organizations are shown in Table-1. At both the organizations, a significant number of users (68% and 77%) used execute-only permissions on their home directories.

Org.	# Users	# ReadX	# NoPerms	# X-only
Org-1	836	198	54	573
Org-2	768	136	39	593

Table 1: Case Study Organization Characteristics. # ReadX is the number of users with read and execute permissions to their home directories, # NoPerms are users with no permissions and # X-only are the users with only execute permissions

Org.	# Hit Users	# Hits	# Files	Data Size
Org-1	462	2409	983086	82 GB
Org-2	380	911	364932	25 GB

Table 2: Data extracted from X-only home directory permissions. # Hit Users is the number of users that leaked private information. # Hits is the total number of directory name hits against all X-only users. # Files is the number of leaked files and Data-Size is the total size of those files

Table-2 lists the amount of data extracted from execute-only home directories at Org-1 and 2.

As can be seen, a large fraction of users indeed leaked private information - 55% and 49% of total users respectively. Recall that we do not extract any data from users with *read* permissions on their home directories; so a more useful number is the fraction of X-only users that revealed private information. That number is 80% and 64% respectively. Also, on an average, 2127 files and 177 MB of data is leaked in the first organization for each X-only user and 960 files and 65 MB of data is leaked in the second organization. A partial reason for the lower numbers in the second organization could be the fewer number of users with *read* permissions, which would have impacted the global name lists creation. Overall, we believe this to be a very significant privacy breach.

As mentioned earlier, many times the names of the subdirectories do not need to be guessed and can be obtained from the history files in the user home directories. Table-3 lists the success rate of the attack in exploiting history files. As it shows, around 40% of X-only users had readable history files which led to 40-50% of total leaked data in size.

Email Statistics

Table-4 presents the results of the email data extracted from users in both organizations. Recall that this data is obtained for both X-only users and the users with *read* permissions on their home directories.

As can be seen, a large number of emails are accessible to unauthorized users (especially at Org-1). Also, the

Org.	# History Hits	# Files	Data Size
Org-1	253	561254	35 GB
Org-2	237	155826	14 GB

Table 3: Exploiting History Files. # History Hits is the number of users with readable history files. # Files is the number of private files leaked due to directory names obtained from history files and Data-Size is the size of the leaked data

Org.	# Folders	# Emails	Size	# Password
Org-1	2509	315919	4.2 GB	6352
Org-2	505	38206	120 MB	237

Table 4: Email Statistics. # Folders is the number of leaked email folders. # Emails is the total number of leaked emails. Size is the size of leaked data and # Password is the number of times the word “password” or its variants appeared in the emails

number of times the word “password” or its variants appear in these emails is alarming. Even though we understand that some of these occurrences might not be accompanied by actual passwords, by personal experience, distributing passwords via emails is by no means an uncommon event.

Browser Statistics

The second organization did not have the mozilla vulnerability since they had a more recent version of the browser installed, by which time the bug had been corrected. So the results shown in Table-5 have been obtained only from the first organization. Looking at the results, the amount of accessible private information is enormous. Figure-1 contains a sample of the websites that had their passwords extractable and clearly most of these websites are extremely sensitive and a privacy breach of this sort is completely unacceptable.

Note that some obtained passwords were for accounts in other institutions and a few of them are likely to be *nix systems. Thus, it is conceivable that this password

# Users with accessible .mozilla	311 (54%)
# Users with readable password DB	149 (26%)
# Passwords Retrievable	579
# Users with readable cookies DB	207 (36%)
# Cookies Retrievable	19456
# Users with accessible caches	233 (40%)
# Cached Entries	20907
# Users with readable browsing histories	256 (44%)
# URLs in History	130,503

Table 5: Browser Statistics at Org-1

Financial Websites www.paypal.com www.ameritrade.com www.bankofamerica.com	Personal Websites adultfriendfinder.com www.hthstudents.com www.icers911.org
Email Accounts mail.lycos.com my.screenname.aol.com webmail.bellsouth.net	Other Institutions cvpr.cs.toronto.edu e8.cvl.iis.u-tokyo.ac.jp systems.cs.colorado.edu

Figure 1: Sample accounts with retrievable passwords

extraction can be used to **expand to other *nix installations** and thus be much more severe in scope than a single installation.

3.3 Attack Severity

It is important to highlight the severity of this attack:

- **Low Technical Sophistication:** The attack is extremely low-tech; the commands used in a manual attack would be `cd`, `ls` and such. This aspect makes the threat significantly more dangerous than most other vulnerabilities.
- **Low Detection Possibility:** In absence of extensive logging, as typically is the case, this attack has a very low probability of detection and even if detected by means like modified last access time, the attacker can not be identified.
- **No Quick Fix:** Unlike most other security vulnerabilities, this attack uses a **design** shortcoming combined with user/application carelessness and no patches would correct this problem overnight.
- **High Success Rate:** The attack had a high success rate at installations where most users are computer literate. With increasing mainstream penetration of *nix systems, ordinary users cannot be expected to fully understand the vulnerabilities. This makes this attack a very potent threat.

4 Privacy Enhancements

The results presented in the previous section clearly establish the fact that there needs to be much better privacy protection in *nix installations. We are currently exploring two enhancement approaches. The first solution is a Privacy Auditing Tool that monitors the privacy health of an organization and can alert users/administrators of potential threats. Analogous to the enterprise security applications that monitor virus and other malicious activity, the privacy auditing tool periodically monitors user home directories and identifies potential data exposures. The second approach is a more proactive approach that

modifies the file system hierarchy by virtualizing it differently for different users. Ensuring the most private user data stays only in the owner's view, many forms of inadvertant exposure can be avoided.

5 Conclusions

In this paper, we critically analyzed the *nix access control model for privacy support in its data sharing mechanisms. We identified a number of design inadequacies that, combined with user or application's privacy-indifferent behavior, lead to privacy breaches. We tested two *nix installations of a few hundred users and found that a massive amount of private data is inadequately protected including emails and actual passwords to financial and other sensitive websites. We briefly proposed two enhancement approaches that can improve data confidentiality in *nix systems and we continue to develop these approaches for our future work.

References

- [1] Mozilla Contents <http://www.holgermetzger.de/pdl.html>.
- [2] Mozilla web browser <http://www.mozilla.org>.
- [3] Opera web browser <http://www.opera.com>.
- [4] D. Cappelli and Michelle Keeney. Insider threat: Real data on a real problem. *USSS/CERT Insider Threat Study, 2004*.
- [5] A. Grunbacher and A. Nuremberg. POSIX Access Control Lists on Linux. <http://www.suse.de/agruen/acl/linux-acls/online>.
- [6] Carnegie Mellon Software Engineering Institute. CERT.
- [7] Linux Manual Pages. *man command-name*.
- [8] Mozilla Bug Report. Bug 59557. <https://bugzilla.mozilla.org/show%5Fbug.cgi?id=59557>.
- [9] Emil Sit and Kevin Fu. Web Cookies: Not Just a Privacy Risk. *Communications of the ACM*, 44(9), 2001.
- [10] United States Secret Service and CERT Coordination Center. E-Crime Watch Survey. 2004.

Notes

¹It is believed that such attacks are usually much under-reported for lack of concrete evidence or fear of negative publicity [10]

²Due to the sensitive nature of the task (measuring *private* content) we took precautions to ensure that our study does *not* violate user privacy by anonymizing users, randomizing scan orders and only collecting aggregates

Securing Web Service by Automatic Robot Detection

KyoungSoo Park, Vivek S. Pai
Princeton University

Kang-Won Lee, Seraphin Calo
IBM T.J. Watson Research Center

Abstract

Web sites are routinely visited by automated agents known as Web robots, that perform acts ranging from the beneficial, such as indexing for search engines, to the malicious, such as searching for vulnerabilities, attempting to crack passwords, or spamming bulletin boards. Previous work to identify malicious robots has relied on ad-hoc signature matching and has been performed on a per-site basis. As Web robots evolve and diversify, these techniques have not been scaling.

We approach the problem as a special form of the Turing test and defend the system by inferring if the traffic source is human or robot. By extracting the implicit patterns of human Web browsing, we develop simple yet effective algorithms to detect human users. Our experiments with the CoDeeN content distribution network show that 95% of human users are detected within the first 57 requests, and 80% can be identified in only 20 requests, with a maximum false positive rate of 2.4%. In the time that this system has been deployed on CoDeeN, robot-related abuse complaints have dropped by a factor of 10.

1 Introduction

Internet robots (or bots) are automated agents or scripts that perform specific tasks without the continuous involvement of human operators. The enormous growth of the Web has made Internet bots indispensable tools for various tasks, such as crawling Web sites to populate search engines, or performing repetitive tasks such as checking the validity of URL links.

Unfortunately, malicious users also use robots for various tasks, including (1) harnessing hundreds or thousands of compromised machines (zombies) to flood Web sites with distributed denial of service (DDoS) attacks, (2) sending requests with forged referrer headers to automatically create “trackback” links that inflate a site’s search engine rankings, (3) generating automated “click-throughs” on online ads to boost affiliate revenue, (4) harvesting e-mail addresses for future spamming, and (5) testing vulnerabilities in servers, CGI scripts, etc., to compromise machines for other uses.

In this paper, we describe our techniques for automatically identifying human-generated Web traffic and separating it from robot-generated traffic. With this information, we can implement a number of policies, such as rate-limiting robot traffic, providing differentiated ser-

vices, or restricting accesses. Such identification can help protect individual Web sites, reduce the abuse experienced by open proxies, or help identify compromised computers within an organization.

Distinguishing between humans and robots based on their HTTP request streams is fundamentally difficult, and reminiscent of the Turing test. While the general problem may be intractable, we can make some observations that are generally useful. First, we observe that most Web browsers behave similarly, and these patterns can be learned, whereas the behaviors of specialized robots generally deviate from normal browsers. Second, the behavior of human users will be different from robots – for example, human users will only follow visible links, whereas crawlers may blindly follow all the links in a page. Third, most human users browse Web pages using the mouse or keyboard, whereas robots need not generate mouse or keyboard activity. From these observations, we propose two effective algorithms for distinguishing human activities from bots in real time: (1) human activity detection, and (2) standard browser detection.

In our study, we test the effectiveness of these algorithms on live data by instrumenting the CoDeeN open-proxy-based content distribution network [9]. Our experiments show that 95% of human users can be detected within the first 57 requests, 80% can be identified within 20 requests, and the maximum false positive rate is only 2.4%. Four months of running our algorithms in CoDeeN indicates that our solution is effective, and reduces robot-related abuse complaints by a factor of 10.

2 Approach

In this section, we describe how to identify human-originated activity and how to detect the patterns exhibited by Web browsers. We implement these techniques in Web proxy servers for transparency, though they could also be implemented in firewalls, servers, etc. We use the term “server” in the rest of this paper to designate where the detection is performed.

2.1 Human Activity Detection

The key insight behind this technique is that we can infer that human users are behind the Web clients (or browsers) when the server gets the evidence of mouse movement or keyboard typing from the client. We detect this activity by embedding custom JavaScript in the pages served to the client. In particular, we take the following steps:

```

<html>
...
<script language="javascript"
src="./index_0729395150.js"></script>
<body onmousemove="return f();">
<script>
function getuseragnt()
{ var agt = navigator.userAgent.toLowerCase();
  agt = agt.replace(/ /g, "");
  return agt;
}
document.write("<link rel='stylesheet\'"
+ "type='text/css\'"
+ "href=http://www.example.com/"
+ getuseragnt() + ">");
</script>
...
</body>
...
</html>

```

```

<!-- ./index_0729395150.js -->

var do_once = false;
function f()
{
  if (do_once == false) {
    var f_image = new Image();
    do_once = true;
    f_image.src = 'http://www.example.com/0729395160.jpg';
    return true;
  }
  return false;
}

```

Figure 1: Modified HTML and its linked JavaScript code. `<script>...</script>` and “onmousemove=...” is dynamically added in the left HTML. The second `<script>..</script>` sends the client’s browser agent string information back to the server.

1. When a client requests page ‘foo.html’, the server generates a random key, $k \in [0, \dots, 2^{128} - 1]$ and records the tuple `<foo.html, k >` in a table indexed by the client’s IP address. The table holds multiple entries per IP address.
2. The server dynamically modifies ‘foo.html’ and delivers it to the client. It includes JavaScript that has an event handler for mouse movement or key clicks. The event handler fetches a fake embedded object whose URL contains k , such as `http://example.com/foo.html.k.jpg`. Suppose the correct URL is U_0 . To prevent smart robots from guessing U_0 without running the script, we obfuscate the script with additional entries such that it contains $m(>0)$ similar functions that each requests U_1, \dots, U_m , where U_i replaces k with some other random number $k_i (\neq k)$. Adding lexical obfuscation can further increase the difficulty in deciphering the script.
3. When the human user moves the mouse or clicks a key, the event handler is activated to fetch U_0 .
4. The server finds the entry for the client IP, and checks if k in the URL matches. If so, it classifies the session as human. If the k does not match, or if no such requests are made, it is classified as a robot. Any robot that blindly fetches embedded objects will be caught with a probability of $\frac{m-1}{m}$.

Figure 1 shows an example of dynamic HTML modification at `www.example.com`. For clarity of presentation, the JavaScript code is not obfuscated. To prevent caching the JavaScript file at the client browser, the server marks it uncacheable by adding the response header line “Cache-Control: no-cache, no-store”.

The sample code shows the mouse movement event handler installed at the `<body>` tag, but one can use other

tags that can easily trigger the event handler, such as a transparent image map (under the `<area>` tag) that covers the entire display area. Other mouse related events such as “mouseup” and “mousedown” can also be used as well as keyboard events. Alternatively, one can make all the links in the page have a mouse click handler. For example, the following code

```

<A HREF=somelink.html onclick='return f();'>
Follow me</A>

```

will call the function `f()` when a human user clicks the “Follow me” link. The function `f()` contains

```

f_image.src
='http://www.example.com/0729395160.jpg';

```

which has the side effect of fetching the image, so the server will receive the mouse movement evidence with $k = 0729395160$ in the URL. The server can respond with any JPEG image because the picture is not used. The parameter k in the URL prevents replay attacks, so the server should choose k at random for each client/page.

The script below the `<body>` tag in Figure 1 sends the client’s browser string to the server. This embedded JavaScript tells the server whether the client enables JavaScript or not. If the client executes the code, but does not generate the mouse event, the server can infer that it is a robot capable of running the JavaScript code.

2.2 Browser Testing

In practice, a small fraction of users (4 – 6% in our study) disable JavaScript on their browsers for security or other reasons. To avoid penalizing such users, we employ browser detection techniques based on the browsing patterns of common Web browsers. The basic idea behind this scheme is that if the client’s behavioral pattern deviates from that of a typical browser such as IE, Firefox, Mozilla, Safari, Netscape, Opera, etc., we assume

it comes from a robot. This can be considered a simplified version of earlier robot detection techniques [6], and allows us to make decisions on-line at data request rates. Basically, we collect request information within a session and try to determine whether a given request has come from a standard browser. This provides an effective measure without overburdening the server with excessive memory consumption. An obvious candidate for browser detection, the “User-Agent” HTTP request header, is easily forged, and we find that it is commonly forged in practice. As a result, we ignore this field.

On the other hand, we discover that many specialized robots do not download certain embedded objects in the page. Some Web crawlers request only HTML files, as do email address collectors. Referrer spammers and click fraud generators do not even need to care about the content of the requested pages. Of course, there are some exceptions like off-line browsers that download all the possible files for future display, but the goal-oriented robots in general do not download presentation-related information (e.g., cascading style sheet (CSS) files, embedded images), or JavaScript files because they do not need to display the page or execute the code.

We can use this information to dynamically modify objects in pages and track their usage. For example, we can dynamically embed an empty CSS file for each HTML page and observe if the CSS file gets requested.

```
<LINK REL="stylesheet" TYPE="text/css"
  HREF="http://www.example.com/2031464296.css">
```

Since CSS files are only used when rendering pages, this technique can catch many special-purpose robots that ignore presentation-related information. We can also use silent audio files or 1-pixel transparent images for the same purpose. Another related but inverse technique is to place a hidden link in the HTML file that is not visible to human users, and see if the link is fetched.

```
<A HREF="http://www.example.com/hidden.html">
<IMG SRC="http://www.example.com/transp_1x1.jpg">
</A>
```

Because the link is placed on a transparent image which is invisible to human users, humans should not fetch it. However, some crawlers blindly follow all the links, including the invisible ones.

3 Experimental Results

To evaluate the effectiveness of our techniques in a real environment, we have implemented them in the CoDeeN content distribution network [9]. CoDeeN consists of 400+ PlanetLab nodes and handles 20+ million requests per day from around the world. Because CoDeeN nodes look like open proxies, they unintentionally attract many robots that seek to abuse the network using the anonymity

Description	# of Sessions	Percentage(%)
Downloaded CSS	268,952	28.9
Executed JavaScript	251,706	27.1
Mouse movement detected	207,368	22.3
Passed CAPTCHA test	84,924	9.1
Followed hidden links	9,323	1.0
Browser type mismatch	6,288	0.7
Total sessions	929,922	100.0

Table 1: CoDeeN Sessions between 1/6/06 and 1/13/06

provided by the infrastructure. While our previous work on rate limiting and privilege separation [9] prevented much abuse, we had to resort to manual pattern detection and rule generation as robots grew more advanced.

3.1 Results from CoDeeN Experiments

We instrumented the CoDeeN proxies with our mechanisms and collected data during a one-week period (Jan 6 - 13, 2006), with some metrics shown in Table 1. For this analysis, we define a session to be a stream of HTTP requests and responses associated with a unique <IP, User-Agent> pair, that has not been idle for more than an hour. To reduce the noise, we only consider sessions that have sent more than 10 requests.

Of the 929,922 sessions total, 28.9% retrieved the empty CSS files we embedded, indicating that they may have used standard browsers. On the other hand, we have detected mouse movements in 22.3% of the total sessions, indicating that they must have human users behind the IP address. Considering that some users may have disabled JavaScript on their browsers, this 22.3% effectively is a lower bound for human sessions.

We can gain further insight by examining the sessions that have executed the embedded JavaScript, but have not shown any mouse movement – these definitely belong to robots. We can calculate the human session set S_H by:

$$S_H = (S_{CSS} \cup S_{MM}) - (S_{JS} - S_{MM})$$

where S_{CSS} are sessions that downloaded the CSS file, S_{MM} are sessions with mouse movement, and S_{JS} are sessions that executed the embedded JavaScript. We consider the sessions with CSS downloads and mouse movements to belong to human users except the ones that have executed JavaScript without reporting mouse movement. We label all other sessions as belonging to robots. Using the data collected from CoDeeN, we calculate that 225,220 sessions (24.2% of total sessions) belong to S_H .

Note that the above equation gives us an upper bound on the human session set because this set has been obtained by removing from the possible human sessions any that clearly belong to robot sessions. However, the difference between the lower bound (22.3%) and the upper bound (24.2%) is relatively tight, with the maximum false positive rate(# of false positives/# of negatives) = $1.9\%/77.7\% = 2.4\%$.

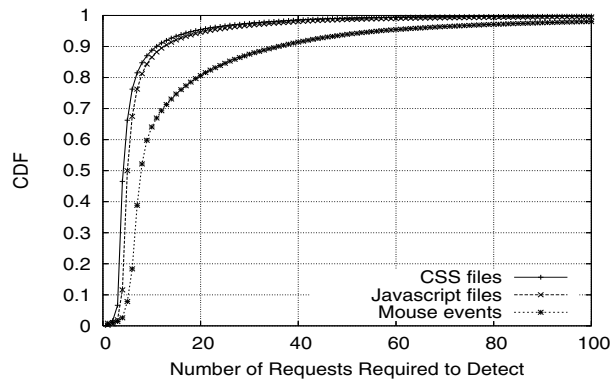


Figure 2: CDF of # of Requests Needed to Detect Humans

Our mouse detection scheme relies on the widespread use of JavaScript among the users. To understand how many users have disabled JavaScript on their browsers, we employed a CAPTCHA test [8, 1] during the data collection period. Users were given the option of solving a CAPTCHA with an incentive of getting higher bandwidth. We see that 9.1% of the total sessions passed the CAPTCHA, and we consider these human users.¹ Of these sessions, 95.8% executed JavaScript, and 99.2% retrieved the CSS file. The difference (3.4%) are users who have disabled JavaScript in their browsers, which is much lower than previously reported numbers ($\sim 10\%$). This can explain the gap between the lower bound and the upper bound of human sessions in our experiment. We also note that most standard browsers request CSS files, suggesting that our algorithm based on CSS file downloads is a good indicator for fast robot detection.

Figure 2 shows how many requests are needed for our schemes to classify a session as human or robot. 80% of the mouse event generating clients could be detected within 20 requests, and 95% of them could be detected within 57 requests. Of clients that downloaded the embedded CSS file, 95% could be classified within 19 requests and 99% in 48 requests. The clients who downloaded JavaScript files show similar characteristics to the CSS file case. Thus, the standard browser testing is a quick method to get results, while human activity detection will provide more accurate results provided a reasonable amount of data. We revisit this issue in Section 4 when we discuss possible machine learning techniques.

3.2 Experience with CoDeeN's Operation

During the four months this scheme has been deployed in CoDeeN, we observed that the number of complaints filed against CoDeeN has decreased significantly. Fig-

¹While some CAPTCHA tests can be solved by character recognition, this one was optional, and active only for a short period. We saw no abuse from clients passing the CAPTCHA test, strongly suggesting they were human.

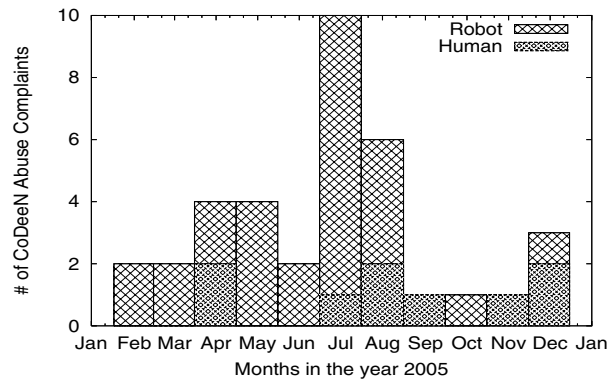


Figure 3: # of CoDeeN Complaints Excluding False Alarms

ure 3 presents the number of complaints filed against the CoDeeN project in 2005. In February, we expanded the deployment of CoDeeN on PlanetLab, from 100 US-only nodes to over 300 total nodes worldwide. As CoDeeN became widely used, the number of complaints started to rise (peaking in July, when most of the complaints were related to referrer spam and click fraud). In late August, we deployed the standard browser test scheme on CoDeeN, and enforced aggressive rate limiting on the robot traffic. After we classify a session to belong to a robot, we further analyzed its behavior (by checking CGI request rate, GET request rate, error response codes, etc.), and blocked its traffic as soon as its behavior deviated from predefined thresholds. After installing this mechanism, we observed the number of complaints related to robot activities have decreased dramatically, to only two instances over four months. During this period, the other complaints were related to hackers, who tried to exploit new PHP or SQL vulnerabilities through CoDeeN. The mouse movement detection mechanism was deployed in January 2006, and we have not received any complaints related to robots as of April 17th.

We also investigate how much additional overhead these schemes impose, and we find it quite acceptable. A fake JavaScript code of size 1KB with simple obfuscation is generated in 144 μ seconds on a machine with a 2 GHz Pentium 4 processor, which would contribute to little additional delay in response. The bandwidth overhead of fake JavaScript and CSS files comprise only 0.3% of CoDeeN's total bandwidth.

4 Discussions and Future Work

In this section, we discuss limitations of our current system and possible improvements using machine learning.

4.1 Limitations of Our Approach

Our proposed detection mechanism is not completely immune to possible countermeasures by the attackers. A serious hacker could implement a bot that could generate

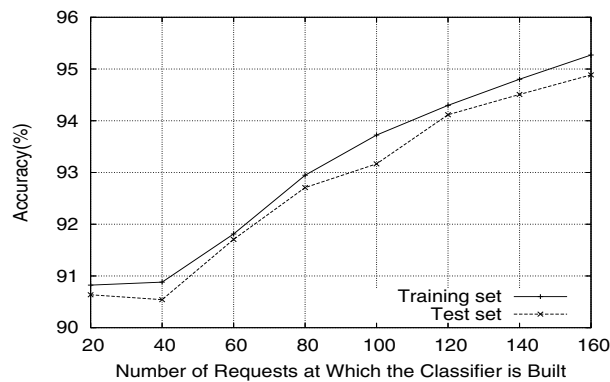


Figure 4: Machine Learning Performance to Detect Robots

mouse or keystroke events if he or she knows that a human activity detection mechanism has been implemented by a site. Although we are not aware of any such intelligent bots today, we may need to address such issues in the future. One possible way to address the problem is to utilize a trusted underlying computer architecture to guarantee that the events have been truly generated by the physical devices [7]. A more practical solution may combine multiple approaches in a staged manner – making quick decisions by fast analysis (e.g., standard browser test), then perform a careful decision algorithm for boundary cases (e.g., AI-based techniques). Our goal in this paper was to design a fast and effective robot detection algorithm that could be deployed in CoDeeN to effect practical benefits, which we seem to have achieved. However, we do not feel the work is complete; on the contrary, it has just started.

4.2 Detection by Machine Learning

From the above discussion, the following question naturally follows: “How effective is a machine learning-based technique, and what is the trade-off?” The main forte of machine learning is that if we can characterize the typical features of human browsing, we can easily detect unwanted traffic by robots. Conceivably, it is very hard to make a bot that behaves exactly like a human.

To test the effectiveness of a detection algorithm using machine learning, we collected data by running CAPTCHA tests on CoDeeN for two weeks, and classified 42,975 human sessions and 124,271 robot sessions using 12 attributes shown in Table 2. We then divided each set into a training set and a test set, using equal numbers of sessions drawn at random. We built eight classifiers at multiples of 20 requests, using the training set. For example, the classifier at the request number 20 means that the classifier is built calculating the attributes of the first 20 requests, and the 40-request classifier uses the first 40 requests, etc. We used AdaBoost [5] with 200 rounds.

Figure 4 shows the accuracy of classification with respect to the number of requests. The result shows the clas-

Attribute	Explanation
HEAD %	% of HEAD commands
HTML %	% of HTML requests
IMAGE %	% of Image(content type=image/*)
CGI %	% of CGI requests
REFERRER %	% of requests with referrer
UNSEEN REFERRER %	% of requests with unvisited referrer
EMBEDDED OBJ %	% of embedded object requests
LINK FOLLOWING %	% of link requests
RESPCODE 2XX %	% of response code 2XX
RESPCODE 3XX %	% of response code 3XX
RESPCODE 4XX %	% of response code 4XX
FAVICON %	% of favicon.ico requests

Table 2: 12 Attributes used in AdaBoost

sification accuracy ranges from 91% to 95% with the test set, and it improves as the classifier sees more requests. From our experiment, RESPCODE 3XX%, REFERRER % and UNSEEN REFERRER % turned out to be the most contributing attributes. Basically, robots do not frequently make requests that result in redirection; many bot requests do not have a valid referrer header field; and finally, referrer spam bots frequently trip the unseen referrer trigger.

Although this approach is promising, it has a few drawbacks. First, it requires significant amount of computation and memory, which may make the server susceptible to DoS attacks. Second, in order to make accurate decisions, it needs a relatively large number of requests, making it difficult to apply in a real-time scenario (in our experiment, it takes 160 requests to achieve 95% accuracy). Third, the human browsing pattern may change with the introduction of a new browser with novel features. Finally, attribute selection must be done carefully. In theory, the learning algorithm can automatically determine the most effective attributes, but in practice, bad choices can decrease the effectiveness of the classifier.

5 Related Work

Despite the potential significance of the problem, there has been relative little research in this area, and much of it is not suitable for detecting the robots in disguise. For example, Web robots are supposed to adhere to the robot exclusion protocol [4], which specifies easily-identified User-Agent fields, with contact information. Before crawling a site, robots should also retrieve a file called “robots.txt”, which contains the access permission of the site defined by the site administrator. Unfortunately, this protocol is entirely advisory, and malicious robots have no incentive to follow it.

Tan *et al.* investigated the navigational pattern of Web robots and applied a machine learning technique to exclude robot traces from the Web access log of a Web site [6]. They note that the navigational pattern of the Web crawlers (e.g., type of pages requested, length of a session, etc.) is different from that of human users, and

these patterns can be used to construct the features to be used by a machine learning algorithm. However, their solution is not adequate for real-time traffic analysis since it requires a relatively large number of requests for accurate detection. Robertson *et al.* tried to reduce administrators' effort in handling the false positives from learning-based anomaly detection by proposing the generalization (deriving anomaly signatures to group similar anomalies) and characterization (to give concrete explanation on the type of the attacks) techniques [3]. Using generalization, one can group similar anomalies together, and can quickly dismiss the whole group in the future if the group belongs to false positives. In contrast to these approaches, our techniques are much simpler to implement yet effective in producing accurate results for incoming requests at real time. Moreover, our proposed solution is robust since it does not have any dependency on specific traffic models or behavior characterizations, which may need to change with the introduction of more sophisticated robots.

CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) is a test consisting of distorted images or sounds, sometimes with instructive description, that are designed to be difficult for robots to decipher [8]. These tests are frequently used by commercial sites which allow only human entrance or limit the number of accesses (e.g. concert ticket purchasing). Kandula *et al.*, used CAPTCHA tests to defend DDoS attacks by compromised robots that mimic the behavior of flash crowds [2]. They optimize the test serving strategy to produce better goodput during the attack. Although CAPTCHA tests are generally regarded as a highly effective mechanism to block robots, it is impractical in our scenario, since human users do not want to solve quiz every time they access a Web page. In comparison, our techniques do not need explicit human interaction, and can be used on every page, while producing highly accurate results. Also we are more concerned in providing a better service under a normal operation rather than special situations such as during denial of service attacks.

6 Conclusion

While Web robots are an important and indispensable part of the modern Web, the subset of malicious robots poses a significant and growing concern for Web sites, proxy networks, and even large organizations. We believe the first step to deal with this is to accurately classify the traffic source, and we present two novel techniques for discriminating humans from robots.

Our experience with CoDeeN shows that our solution is highly effective in identifying humans and robots. 95% of humans are detected within 57 requests with less than a 2.4% false positive rate. The integration of the techniques in CoDeeN's operation also greatly reduced the number of abuse complaints caused by robots. Furthermore, we

believe that our solution is quite general – not only does it apply to the safe deployment of open proxies, but it can be used to identify streams of robot traffic in a wide variety of settings. We believe that this approach can be applied both to individual Web sites, and to large organizations trying to identify compromised machines operating inside their networks.

Acknowledgments

We thank Felix Holderied and Sebastian Wilhelmi [1] for providing the CAPTCHA image library, and Peter Kwan for useful discussion and feedback. This work was supported in part by NSF Grants ANI-0335214, CNS-0439842, and CNS-0520053.

References

- [1] captchas.net.
<http://www.captchas.net/>.
- [2] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, 2005.
- [3] W. Robertson, G. Vigna, C. Krugel, and R. A. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proceedings of The 13th Annual Network and Distributed System Security Symposium (NDSS '06)*, 2006.
- [4] Robot Exclusion Protocol.
<http://www.robotstxt.org/wc/exclusion.html>.
- [5] R. Schapire. The boosting approach to machine learning: An overview. *Nonlinear Estimation and Classification*, 2003.
- [6] P.-N. Tan and V. Kumar. Discovery of web robot sessions based on their navigational patterns. *Data Mining and Knowledge Discovery*, 6:9–35, 2002.
- [7] Trusted Computing Group.
<http://www.trustedcomputinggroup.org/>.
- [8] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *Proceedings of Eurocrypt*, pages 294–311, 2003.
- [9] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

Cutting through the Confusion: A Measurement Study of Homograph Attacks

Tobias Holgers, David E. Watson, and Steven D. Gribble
Department of Computer Science & Engineering
University of Washington

1 Introduction

Domain names are crucial to the usability of the Web, but the same characteristics that make them useful to people also make them vulnerable to attack. When a user follows a hyperlink, the domain name within the URL provides her with the first and most important indication of the identity of the organization with which she will interact. If the user is fooled into misreading a domain name, she will believe she is interacting with one organization, but she might actually be interacting with an attacker. By spoofing the content of the user's intended destination, the attacker might trick the user into revealing sensitive information. In this scenario, SSL is no help to the victim, since the attacker could obtain a valid certificate for the confused domain name.

A *homograph attack* is one technique for carrying out this scheme. A homograph is a letter or string that is visually confusable with a different letter or string. For example, using most sans-serif fonts, the Latin letter l (lower case 'el') is visually confusable with the Latin letter I (upper case 'eye'). Rendered with such a font, the following are *confusable*, if not indistinguishable:

`http://www.paypal.com` vs. `http://www.paypai.com`

An attacker who registers the confusable domain name `paypai.com` therefore may be able to lure victims to their site, for example by sending spam that appears to contain a hyperlink to the *authoritative* PayPal site.

Web homograph attacks have existed for some time, and the recent adoption of International Domain Names (IDNs) support by browsers and DNS registrars has exacerbated the problem [Gabr02]. Many international letters have similar glyphs, such as the Cyrillic letter p (lower case 'er,' Unicode 0x0440) and the Latin letter p. Because of the large potential for misuse of IDNs, browser vendors, policy advocates, and researchers have been exploring techniques for mitigating homograph attacks [Mozi05, Appl05, Oper05, Mark05].

There has been plenty of attention on the problem recently, but we are not aware of any data that quantifies the degree to which Web homograph attacks are currently taking place. In this paper, we use a combination of passive network tracing and active DNS probing to measure

several aspects of Web homographs. Our main findings are four-fold.

First, many authoritative Web sites that users visit have several confusable domain names registered. Popular Web sites are much more likely to have such confusable domains registered. Second, registered confusable domain names tend to consist of single character substitutions from their authoritative domains, though we saw instances of five-character substitutions. Most confusables currently use Latin character homographs, but we did find a non-trivial number of IDN homographs. Third, Web sites associated with non-authoritative confusable domains most commonly show users advertisements. Less common functions include redirecting victims to competitor sites and spoofing the content of authoritative site. Fourth, during our nine-day trace, none of the 828 Web clients we observed visited a non-authoritative confusable Web site.

Overall, our measurement results suggest that homograph attacks currently are rare and not severe in nature. However, given the recent increases in phishing incidents, homograph attacks seem like an attractive future method for attackers to lure users to spoofed sites.

2 Homographs and Confusability

As previously mentioned, a homograph is a letter or string that has enough of a visual similarity to a different letter or string that the two may be confused for one another. The precise degree of similarity necessary to cause confusion is difficult to quantify, as it depends on the observer, the fonts and font sizes used, and the context in which the homograph is observed.

There are many different categories of confusable characters. They may be drawn from the same script, such as the Latin characters '-' (hyphen) and '–' (en dash). Different scripts may be involved, such as with the Latin character a and the Cyrillic character a (small letter a). Font choices can affect confusability; the Latin characters 'rn,' if rendered with a sans-serif font appear as 'rn' and can be confused with the Latin character 'm.' Two characters with very different glyphs may appear to be identical if a browser does not have support for one of

them. For example, an ä ('a' with an umlaut) might be rendered without the umlaut.

Further compounding the problem is the fact that confusable characters do not need to be used when constructing confusable strings. The word *recieve* may be confused with *receive*, and even more complicated misspellings may be overlooked by a causal observer. Given all of this complexity, in this paper we do not attempt to establish perceptual thresholds of confusability and rigorously examine all possible confusable characters. Instead, we make the simplifying assumption that two characters are confusable if and only if they are listed as confusable in the Unicode Technical Report on security considerations [Davi05].

This assumption gives us only a rough approximation to the real world notion of confusability, however, as we will show in Section 3, many registered domain names do have confusable domains registered consisting of character substitutions. In most cases, these confusable domains do not have a legitimate purpose.

With this assumption in place, we can operationally define the confusability of two strings: one string is confusable with another string if and only if they are related by some number of confusable character substitutions. As an example, consider the following string, which contains four character substitutions:

Microsoft Corporation

The underlined characters are Cyrillic confusables of their Latin character counterparts. For this particular string, the set of all confusable strings related to it is enormous (32,459,975,614,080), since most of the characters in the string have at least one confusable character associated with them, and we must consider all possible one, two, three, ..., twenty-one character substitutions.

Increasing the number of confusable character substitutions in a string tends to make the string less confusable. Accordingly, in practice confusable strings tend to contain only one or two substitutions, though as we will show in Section 3, some popular domains have registered confusables with up to five character substitutions.

In this paper, we examine a simple kind of homograph attack, in which an attacker registers a domain name that is confusable with some other domain name, presumably to lure victims to their site. In principle, two registered domains may both be associated with legitimate organizations, yet still be confusable with each other. In practice, a given set of confusable domain names tends to consist of a single *authoritative* domain, and a collection of non-authoritative, illegitimate domains. Though authoritativeness is a subjectively defined characteristic, we have found in all cases it is simple to distinguish between the authoritative domain that people intend to visit, and the non-authoritative confusables that attackers create.

3 Measurement Study

We gathered a nine-day-long trace of the Web activity generated by the population of clients in the Department of Computer Science and Engineering at the University of Washington. The department consists of approximately 40 faculty, 40 staff, 275 graduate students, and 450 undergraduate students.

There is a mixture of static IP assignment and DHCP usage in the department, but the majority of hosts that rely on DHCP receive the same IP address in practice. Accordingly, the number of IP addresses we observed in the trace, 828, is a reasonable (though not perfect) estimate of the number of hosts that were active during the trace period.

We installed a passive network tap on the router connecting the departmental subnets to the campus backbone. This tap allowed us to observe all packets flowing between department computers and external hosts. The peak traffic rate through the router was low enough that our network monitoring host dropped no packets.

Using Snort, we collected a trace consisting of all outbound HTTP GET requests. We post-processed the trace to extract the domain name associated with each request. To perform this extraction, we looked in the "Host" HTTP header field; this field is required in HTTP/1.1, and is generated by all modern browsers. Using this field saved us from having to perform reverse DNS lookups, and it also allowed us to disambiguate between multiple domains hosted on the same IP address.

Given this list of domain names, we calculated the popularity of a domain name by counting the number of GET requests directed to it. To transform our object-related popularity measure into an approximate page-relative popularity measure, we excluded requests for image data types, since otherwise a single page containing many embedded images would have a higher contribution to domain popularity than a single page containing few embedded images.

It is clear that the Web activity of a computer science department is not wholly representative of Internet-wide Web activity. However, the set of *popular* Web sites within the departmental trace has a substantial overlap with the set of top 500 global Web properties listed by Alexa Internet [Alex05]: 31 of the top 50 domains in the Alexa list appeared in our trace. As we will show in Section 3.2.2, popular Web sites are more likely to have confusable domain names registered.

3.1 Active DNS probing

Once we obtained the list of domain names from the departmental trace, our next step was to search for registered confusable domain names associated with each

rank	authoritative domain name	# possible confusables	# registered confusables	confusable names (confusable characters underlined, IDN surrogates in parenthesis)
1	yahoo.com	3,202	2	y ^g ahoo.com (xn--yhaoo-5ld.com), yah ^g oo.com
2	netflix.com	12	1	n ^g etflix.com (xn--netflix-com)
3	google.com	1,158	4	g ^q oogle.com, go ^q o ^q le.com, g ^q oo ^q le.com, go ^q o ^q le.com
6	passport.net	19,591	1	pa ^q sspo ^q rt.net
8	ebay.com	252	2	g ^q ebay.com (xn--bay-qdd.com), g ^q eb ^q y.com (xn--by-7kus.com)
11	microsoft.com	46,552	5	m ^q icr ^q osoft.com (xn--micosoft-qdt.com), m ^q icr ^q osoft.com (xn--micosoft- stb.com), m ^q icr ^q osoft.com (xn--micosoft-djg.com), m ^q icr ^q osoft.com (xn-- mif-65ca6f.com), m ^q icr ^q osoft.com
12	amazon.com	3,672	1	am ^q azon.com (xn--amazon-mys.com)
18	facebook.com	1,344	0	
20	zol.com	204	2	zo ^q l.com (xn--ol-jbc.com), zo ^q l.com (xn--ol-fm.com)
22	go.com	17	0	
102	bankofamerica.com	25,909,632	1	ba ^q nkofamerica.com (xn--bnkofamerica-x9j.com)
980	paypal.com	3,455	4	pa ^q ypal.com (xn--pypal-4wc.com), pa ^q ypal.com (xn--napal-7ze.com), pa ^q ypal.com (xn--pypal-7ve.com), pa ^q ypal.com (xn--pysa-53c1n.com)

Table 1: Registered confusables for popular domains. This table lists the registered confusable domains for the 10 most popular English language Web sites within the Alexa 500 list, as well as two financial sites.

one. To accomplish this, for each traced name, we generated confusable names by substituting one or more characters with corresponding confusable characters. Then, we performed a DNS lookup on each generated name to test whether it was actually registered.

There is a combinatorial explosion in the number of confusable names associated with a given string when performing multiple character substitutions. Because of this, we limited our search to confusable names with at most three confusable characters. However, to explore the degree to which this caused us to miss registered confusables with a greater number of substitutions, we performed an exhaustive search of the full space for a few of the traced domains for which we found the most registered confusable names.

Since the deployment trace may be biased towards university and research topics, we conducted a similar evaluation using the list of the Top 500 most popular domain names, according to Alexa [Alex06]. The Alexa list contains domains ordered by a “traffic rank.” This metric is the geometric mean of reach (percent of Internet users visiting the site) and page views (percentage of all daily global page views).

3.2 Results

In Table 2, we show high-level results from our study. We observed 828 clients accessing 3,425 different Web server domain names, issuing a total of 452,664 HTTP GET requests. Web sites visited in our trace were authoritative: no client ever visited a Web site with a non-authoritative, confusable domain name. However,

trace period	June 6 -- June 13, 2006
# client IPs	828
# GET requests	452,664
# server IPs	4,991
# distinctive Web domain names	3,425
# non-authoritative, confusable domain names visited by users	0
# non-authoritative, registered confusable domain names identified by DNS probing	UW trace: 237 Alexa top 500 list: 162
# authoritative sites that have non-authoritative, registered confusables	UW trace: 182 Alexa top 500 list: 116

Table 2: Overall results. This table provides summary statistics describing our trace.

our DNS probing found 399 registered domains whose names are confusable with authoritative Web domains visited by our users. Looking at this data another way, 298 authoritative Web domains have one or more non-authoritative, confusable, registered domains. None of our users appeared to have fallen victim to a homograph attack during our trace period, even though the potential for such an attack does exist.

For those authoritative domains that had confusable domains registered, we typically found a very small number of registered confusable names. Even though a large number of confusable names are possible for a given authoritative domain name, there are usually just a handful of confusable domains registered.

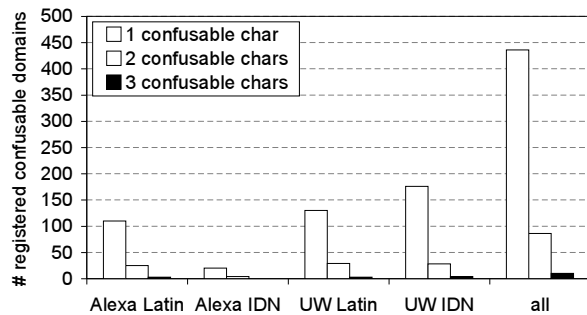


Figure 1: **# confusable character substitutions.** This graph shows how many registered confusables have one, two, or three confusable character substitutions.

In Table 1, we show a list of registered confusable domains found for the top 10 most popular English language Web sites within the Alexa 500 list, as well as two financial sites. Note that this table only reports on registered DNS names with three or fewer confusable character substitutions, as previously described in Section 3.1.

3.2.1 Number of character substitutions

Intuitively, one should expect that registered confusable domain names will tend to consist of a small number of confusable character substitutions. Each confusable character may not always render identically to the intended character. Accordingly, while one confusable character in a confusable domain name may escape notice, two or three such characters may not.

Figure 1 shows that most registered confusable domain names only contain a single confusable character, suggesting this intuition is correct. As well, this data validates our choice of limiting the search space of our DNS probes to names with no more than three character substitutions: less than 3% of confusable names we found had three substitutions.

To further validate this choice, we performed an exhaustive search for confusables using the two domain names with the most registered confusables, microsoft.com and paypal.com. This full search of all 48,552 possible microsoft confusables and 3,456 paypal confusables found only one confusable domain that our limited search missed: a microsoft.com confusable with five confusable character substitutions.

3.2.2 Popularity and registered confusables

Figure 2 shows, for an authoritative site of a given popularity rank, the fraction of all registered confusable names found that are associated with authoritative sites of equal or greater popularity. As well, the figure includes a logarithmic curve fit for the “UW IDN” data

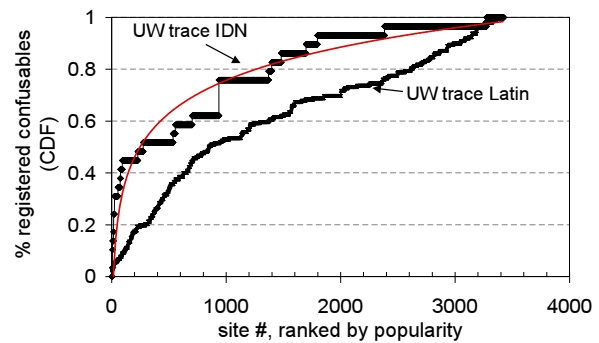


Figure 2: **Popularity vs. registered confusables.** This CDF shows, for a site of a given popularity, the fraction of registered confusable names found that are associated with authoritative sites of equal or greater popularity. Popular sites have more registered confusable names.

series. The graphs show that popular authoritative sites have more registered confusable names than unpopular authoritative sites.

If registered confusable domain names were uniformly distributed across authoritative sites, these lines would have a constant slope. Instead, we see that for both UW IDN and UW Latin confusables, popular authoritative sites have more confusable names registered for them than unpopular authoritative sites. This effect is most striking for IDN confusables; 80% of registered IDN confusables found are associated with the top 30% of authoritative sites. The effect is less striking for Latin confusables, but we hypothesize that the effect would reveal itself more prominently with a longer trace that would include additional unpopular domains.

3.2.3 Latin vs. IDN names

Our search for registered confusable domain names included domains consisting entirely of Latin character substitutions, and IDN domains that included some Unicode character substitutions. In Table 3, we show how many of each of these exist for both the Alexa 500 list and domains visited in the UW trace.

Our results show that most registered confusable domains consist entirely of Latin characters: IDN confusable domains containing Unicode characters account for only 15% and 12% of the Alexa and UW lists, respectively. While a relatively small fraction, IDN confusable domains do have a noticeable presence, and they can be expected to grow as browser support for IDN increases. For example, the upcoming Microsoft Internet Explorer version 7 browser is expected to have IDN support, making confusable Unicode domain names potentially more attractive to attackers.

	Latin	IDN	total
Alexa Top 500	138 (85%)	24 (15%)	162
UW Trace	208 (88%)	29 (12%)	237

Table 3: **Latin vs. Unicode confusables.** This table shows the number of registered confusable domains found that contain only Latin confusable characters, and the number of IDN domains that contain some Unicode confusable characters.

3.2.4 The intent behind confusable domains

Our data shows that many non-authoritative, confusable domain names have been registered. We now turn our attention to understanding what goal attackers had when registering them. Homographs can be used to construct elaborate Web spoofing or phishing attacks, in which the victim is fooled into revealing sensitive information. However, attackers may have other less dangerous goals in mind, such as attracting victims to a site in order to display advertisements.

To understand the attacker's intent behind a confusable domain, and to gauge the current risk that homograph attacks pose, we manually examined all non-authoritative confusable domains that we found registered. Based on our examination, we categorized each site into one of the following seven categories in decreasing order of (subjectively assigned) risk to the victim:

- **Web spoofing:** the confusable site spoofs the content of the authoritative site.
- **Redirect to competitor:** the victim is redirected to a commercial competitor of the authoritative site.
- **Advertisement:** ads are shown to the victim.
- **For sale:** the registered confusable domain name is advertised as being for sale.
- **Unrelated:** the site has content which is unrelated to the authoritative site.
- **No content:** the registered confusable domain name does not have an active Web server, or the server returns blank pages.
- **Redirect to authoritative:** the victim is redirected to a the authoritative site, perhaps as a defensive measure put in place by the authoritative site itself.

A given site may belong in more than one category, such as a site that is for sale and also shows ads. We attempted to emphasize the more subtle, and thus potentially more dangerous, uses of homographs and thus categorized each site only in its highest risk category.

Table 4 summarizes the results. Advertising, a relatively benign function, was overwhelmingly the most

Intent	% of registered confusable domains		
	Alexa 500 list	UW trace	union
Web spoofing	0.6%	1.7%	1.3%
Redirect to competitor	2.5%	2.2%	2.3%
Advertisement	43.2%	45.5%	44.5%
For sale	16.7%	14.3%	13.0%
Unrelated	13.6%	10.8%	12.0%
No content	19.1%	18.2%	20.6%
Redirect to authoritative	4.3%	7.4%	6.3%

Table 4: **Intent of registered confusables.** This table shows the fraction of registered confusable domains that were observed to have the listed intent.

popular use for confusable domain names. There were very few spoofed sites among registered domains we observed. Additionally, we verified that none of these spoofed sites attempted to trick the user into submitting sensitive information. Instead, these spoofed sites either consisted of parodies of the authoritative site, or they served to warn potential victims about the dangers of homograph attacks.

4 Related work

Web spoofing attacks were first considered by [Felt97]. [Gabr02] first discussed using homographs as a part of a web spoofing attack. Early versions of the attack relied on similarities between Latin letters and numbers. For example, an attacker could register an address where *o* is replaced by 0 (zero), or 1 with 1 (one).

With the introduction of International Domain Names (IDN) the number of visually confusable characters has increased dramatically. IDN attacks have been possible in Mozilla [Mozi05], Safari [Appl05] and Opera [Oper05] for at least one publicly available release, though the latest versions have adopted some defensive mechanisms. Browser-based solutions to the homograph problem are currently incomplete, however, as they either rely on trusted registrars or disable significant portions of the IDN namespace.

Registrars issuing IDN domains have been asked to put in place policies to prevent two homographic domains from being registered to different sites [Mark05]. Relying on registrars to help solve the problem has disadvantages, since registrars must contend with multiple jurisdictions and potentially conflicting regulatory restrictions. However, this approach is compatible with other solutions to the Web spoofing problem. For example, trust bars [Herz04], the eBay Toolbar [eBay], and Spoof-

Guard [Chou04] give users immediate and unforgeable security context information.

[Goth05] evaluates the current rate and cost of phishing scams, and concludes that while the cost has been reduced in recent years, it is still costing billions of dollars. [Weny05] discusses using Web crawlers to look for visually similar Web pages. Others researchers in the usability, cryptography, and anti-phishing communities have proposed several mechanisms to defend against phishing attacks. For example, Jakobsson [Jako05] proposes an economic analysis to quantify the risks of an attack and to develop methods for defending against them. As another example, Adida et al. propose the adoption of identity-based ring signatures to provide digitally signed email to eliminate spam-based phishing attacks [Adid05]. Dhamija and Tygar propose the concept of “security skins,” a browser extension that allows remote sites to prove its identity to users in a way that is usable but hard for attackers to spoof.

5 Conclusions

While visually confusable, non-authoritative domains have been registered in practice, the threat actually posed by these domains currently does not live up to the potential feared by the community [Oper05, Mozi05, Appl05]. Many popular Web sites do have associated confusable domains registered, but the most common functions of these confusable domains are benign, such as serving advertisements. However, as support for IDN names grows, homograph attacks do have the potential to become more common and malicious.

Overall, our results show that: (1) users often visited sites that have confusable domains registered, but no user visited one of these non-authoritative domains during our trace; (2) popular sites are much more likely to have registered non-authoritative confusable domains than unpopular sites; (3) confusable domains tend to have a single confusable character within them, and currently only 12-15% of confusable domains rely on Unicode confusable characters; and (4) most confusable domains have relatively benign intent, such as showing advertisements. Though a small fraction do spoof the authoritative site, even these spoofed sites appear to have relatively benign intent, such as parody.

Acknowledgments

This work was supported in part by the National Science Foundation under grants CNS-0430477 and ANI-0132817, by an Alfred P. Sloan Foundation Fellowship, and by gifts from Intel Corporation and Nortel Networks.

References

- [Adid05] Ben Adida, Susan Hohenberger and Ronald L. Rivest, *Separable Identity-based Ring Signatures: Theoretical Foundations for Fighting Phishing Attacks*. DIMACS Workshop on Theft in E-Commerce, Piscataway, New Jersey, April 2005.
- [Alex05] Alexa Web Search, Alexa Internet Inc., *Global Top 500 Sites*, June 7 2005.
- [Appl05] Anonymous, *About Safari International Domain Name support*, Apple Computer Inc., March 2005, <http://docs.info.apple.com/article.html?artnum=301116>
- [Chou04] Neil Chou, Robert Ledesma, Yuka Teraguchi, Dan Boneh and John C. Mitchell, *Client-side defense against web-based identity theft*. Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04), San Diego, CA, February 2004.
- [Davi05] Mark Davis, Draft Unicode Technical Report #36, *Security Considerations in the Implementation of Unicode and Related Technology*, February 20, 2005.
- [Dham05] Rachna Dhamija and J.D. Tygar, *The Battle Against Phishing: Dynamic Security Skins*. Proceedings of the 2005 ACM Symposium on Usable Security and Privacy, July 2005.
- [eBay] eBay Toolbar, available at http://pages.ebay.com/ebay_toolbar.
- [Felt97] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach, *Web Spoofing: An Internet Con Game*, Technical Report 540-96, Department of Computer Science, Princeton University, February 1997.
- [Gabr02] E. Gabrilovich, A. Gontmakher, *The Homograph Attack described*, Communications of the ACM, 45(2):128, February 2002
- [Goth05] Greg Goth, *Phishing Attacks Rising, But Dollar Losses Down*, IEEE Security and Privacy, Volume 3 Issue 1, January 2005
- [Herz04] A. Herzberg, A. Gbara, *TrustBar: Protecting (even Naïve) Web Users from Spoofing and Phishing Attacks*, Bar Ilan University, 2004
- [Jako05] Markus Jakobsson, *Modeling and Preventing Phishing Attacks*, Financial Cryptography 2005
- [Mark05] Gervase Markham, *IDN Update*, March 24, 2005, <http://weblogs.mozillazine.org/gerv/archives/007785.html>
- [Mozi05] Anonymous, *Mozilla Foundation Security Advisory 2005-29*, Mozilla Organization, February 17th, 2005.
- [Oper05] Anonymous, *Advisory: Internationalized domain names (IDN) can be used for spoofing*, Opera Software ASA, February 25th, 2005.
- [Veri05] VeriSign, Inc., *i-Nav Internationalized Domain Name browser plug-in*, <http://www.idnnow.com/index.jsp>
- [Weny05] Liu Wenyin, Guanglin Huang, Liu Xiaoyue, Zhang Min, Xiaotie Deng, *Detection of phishing webpages based on visual similarity*, Special interest tracks and posters of the 14th International conference on World Wide Web, May 2005.

Stealth Probing: Efficient Data-Plane Security for IP Routing

Ioannis Avramopoulos and Jennifer Rexford
Princeton University
{iavramop, jrex}@princeton.edu

Abstract

IP routing is notoriously vulnerable to accidental misconfiguration and malicious attack. Although secure routing protocols are an important defense, the data plane must be part of any complete solution. Existing proposals for secure (link-level) forwarding are heavy-weight, requiring cryptographic operations at each hop in a path. Instead, we propose a light-weight data-plane mechanism (called *stealth probing*) that monitors the availability of paths in a secure fashion, while enabling the management plane to home in on the location of adversaries by combining the results of probes from different vantage points (called *Byzantine tomography*). We illustrate how stealth probing and Byzantine tomography can be applied in today's routing architecture, without requiring support from end hosts or internal routers.

1 Introduction

Most research and standards activity in secure IP routing has focused on the routing protocols, rather than the forwarding of data packets. In this paper, we propose an *operationally viable* approach to providing data-plane security. As an example of the threats we address, consider an attacker that breaks into one or more routers, or a disgruntled network operator with easy access to the routers. The adversary can easily create an unnoticed disruption by installing access control lists (ACLs) that selectively discard data traffic, while leaving the routing protocol intact and allowing probe traffic through. Since the routing protocol does not verify the operation of the data plane, and because the probes are delivered successfully, this attack is extremely difficult to diagnose.

Threat Model: We consider a network where a subset of the routers and links (unknown to the defending entity) are controlled by an adversary. Using these routers and links, the adversary can eavesdrop on host-to-host communications, tamper with the packet contents, im-

personate host services, misdirect network traffic, or render the packet-delivery service unavailable by means of routing-protocol and data-plane attacks. A remote adversary may also deplete data-plane resources through denial-of-service attacks; because other techniques, such as fair queuing and packet filtering, can protect from these attacks, we do not consider them further.

Our primary goal is to secure routing through *data-plane countermeasures* that detect routing-protocol and data-plane attacks that disrupt packet delivery, assuming arbitrary (or *Byzantine*) behavior by the adversary. Ensuring the availability of the network despite the presence of adversaries prevents financial losses and other detrimental societal impacts that these adversaries could otherwise inflict. We do not prevent traffic misdirection attacks (and, thus, we avoid any associated overheads such as cryptographically enforcing a route), though we use network encryption of the data traffic to counter their consequences. Encryption, furthermore, safeguards against eavesdropping, the tampering of packet contents, and the impersonation of host services.

Layering of Countermeasures: In an operationally viable secured routing system, we argue that we should consider all three dimensions of IP routing, i.e., the *data*, *control*, and *management planes*: The data plane supports packet-forwarding functionality, such as destination-based forwarding, filtering, and tunneling. The control plane implements the routing protocols that discover the topology and select routes. The management plane monitors the network and configures the routers. The management and control planes have been the focus of most countermeasures to Byzantine failures.

Management Plane: As a management-plane countermeasure, operators can apply Best Common Practices (BCPs) for securing their infrastructure and filtering suspicious route announcements. These BCPs reduce the likelihood of attacks but cannot prevent them entirely; in addition, an end-to-end path often traverses multiple networks, including some that do not apply BCPs.

Control Plane: Secure routing protocols [8, 10, 14, 18] ensure that valid routing advertisements correctly identify the links between non-faulty routers (or Autonomous Systems). However, these protocols do not prevent false announcements that faulty routers are connected to other faulty routers, as in collusion (or wormhole) attacks [8, 18]. Wormholes along with malicious ACLs can create invisible black holes for the data traffic. Because routing protocols do not verify forwarding behavior, even if perfectly secure routing protocols were deployed, availability could still be compromised. This risk can be mitigated by incorporating secure forwarding functionality in the routing system.

Data Plane: Secure forwarding protocols such as [2, 4] and the Π_2 protocol in [13] provide availability monitoring and secure fault localization at the *link level*. The fine granularity leads to high overhead and complexity (e.g., path-specific authentication and, in certain cases [2, 4], the distribution of pairwise router keys) inappropriate for a generic forwarding paradigm. Although useful for failure recovery, fault localization should not overburden the data plane. We advocate that availability monitoring and fault localization should be cleanly separated, into the data plane (*stealth probing*) and management plane (*Byzantine tomography*), respectively.

Stealth Probing: The stealth-probing protocol we propose in this paper is a data-plane availability monitor. It determines whether a router-to-router path is operational, even if an adversary controls intermediate routers and tries to evade detection. Stealth probing creates an encrypted tunnel between two end-routers and diverts both the data and probe traffic into the tunnel. Since the data and probe packets are indistinguishable, the adversary cannot drop data packets without dropping the probes as well, making it difficult to evade detection. Rather than requiring ubiquitous deployment, stealth probing could be deployed “as needed” to protect critical traffic between selected edge networks.

Stealth probing offers several key practical advantages. First, stealth probing is *incrementally deployable*. Because of its end-router-to-end-router design, stealth probing does not require support from legacy routers in the core of an ISP network or intermediate ASes in an interdomain path. Networks that adopt stealth probing will see immediate benefits even in limited deployment scenarios. Second, stealth probing is *backward compatible* with the existing infrastructure, since the tunnels do not require any support from the internal routers. Finally, stealth probing is *incentive compatible*. Service providers can use the encrypted tunnels to provide other value-added services, such as secure Virtual Private Networks (VPNs), to customers. Encrypted tunnels also protect users from a broader range of attacks such as eavesdropping, tampering, traffic analysis, and misdirection.

2 Stealth Probing

Stealth probing is a secure data-plane monitoring tool that relies on the efficient symmetric cryptographic protection of the IPsec protocol suite, applied in an end-router-to-end-router fashion. In this section, we first discuss the limitations of other approaches to secure data-plane monitoring, followed by an overview of stealth probing. Then, we describe how stealth probing works in greater detail.

2.1 Limitations of Strawman Designs

Stealth probing addresses the problem of securely deciding whether a node-to-node path correctly delivers data packets from one end of the path to the other. An adversary that is present at one or more intermediate nodes of the path must not be able to coerce a false decision. Furthermore, the overhead of the decision process must be practical for deployment in operational networks.

Consider two routers u and v . Let's assume for simplicity that u is a source of data traffic for which v is a sink. We want to verify that this traffic is flowing properly in the forward $u \rightarrow v$ direction.

Probing One approach to meet our objective is for router u to send to v one or more ICMP echo requests and infer the fate of data traffic based on the receipt of ICMP echo replies. This method is non-intrusive since it reaches a decision with a small number of probes. However, if an adversary is present in the path between u and v , he can selectively drop data packets and avoid detection by selectively forwarding echo requests and replies.

Cumulative network-layer ACKs In a second approach, v explicitly acknowledges receipt of a bundle of data packets from u by a cumulative ACK that contains a count of the received packets. However, if an adversary is present in the path between u and v , he can drop data packets and avoid detection by forging destination ACKs. So, let's further assume that u and v share a secret key. Using this key, we can prevent this attack by requiring u to authenticate data packets by means of a message authentication code (MAC) and v to authenticate ACKs in the same way. However, packet counts are insufficient to determine the timeliness of data delivery and, therefore, this scheme is vulnerable to an adversary that *delays* packets. Furthermore, packet counts are insufficient to detect *selective* attacks that target individual IP addresses (or prefixes).

Transport-layer ACKs A third approach is to use a secure (host-to-host) transport layer protocol such as TLS (Transport Layer Security) [5]. However, because this scheme cannot differentiate between host and router failures, it would suffer from “false alarms” due to host failures that would complicate fault localization by the

management plane.

Traceroute A fourth approach is that adopted by traceroute that uses ICMP “time exceeded” and “port unreachable” messages to either determine the full path from a source to a destination or identify the last router before a black hole. Traceroute has fine *link-level* detection granularity but cannot prevent the preferential treatment of its packets by an adversary who can in this way avoid detection. In addition, many ISPs disable their routers from sending ICMP response messages.

2.2 Minimal Secure Data Plane Monitor

Stealth probing has a “minimalist” design: It enables recovery from routing attacks and misconfigurations by offering secure *path-level* failure detection capability, keeping the data-plane support to a minimum. The idea in stealth probing is to use probes to reach a secure decision on the fate of data traffic by establishing an encrypted and authenticated *tunnel* between two routers in the traffic’s path and diverting both the data traffic and the probes into this tunnel. Encryption conceals probing traffic so that it is indistinguishable from data traffic, and authentication makes the tampering of data traffic detectable. Probing can be either *active* or *passive*:

- Active probing uses ICMP echo requests and replies. The size of echo requests is concealed using padding to decrease the number of distinct data-packet sizes. Echo request sizes are then chosen to match data-packet sizes, and inter-probing intervals are randomly jittered.
- In passive probing, the tunnel entry and exit points agree on an efficient (non-cryptographic) hash function to be applied on the immutable fields of each packet—before encryption at the entry point and after decryption at the exit point. If the image of the hash is less than an agreed-upon value, the corresponding data packet serves as an implicit probe that the tunnel exit point must acknowledge. This method is akin to *trajectory sampling* [6]; a Bloom filter may be used to compress the ACKs, similar to how hashes are compressed in [17].

Stealth probing has the following primary benefits:

- Because stealth probing is an *end-router-to-end-router failure detection mechanism*, intermediate routers of a monitored path do not need to explicitly support the stealth probe. Therefore, stealth probing can be deployed across legacy routers and over interdomain paths.
- Stealth probing is *non-intrusive*. The processing requirements at tunnel endpoints (outlined in Section 2.3) are simple and the probing overhead is

minimal. Intermediate routers do not process tunneled packets as they are tunnel agnostic.

- By measuring the round-trip-times of probing traffic, attacks that delay packets are detectable.
- By hiding the source and destination IP addresses of the data traffic, encryption prevents attacks that target individual IP addresses.
- By making the TCP mechanism *opaque*, encryption mitigates attacks that exploit the TCP mechanism.
- The use of tunnels permits *selectivity* in the traffic that is protected. The management plane can configure packet classifiers that identify the critical traffic and direct only the matching packets into the encrypted tunnels.

Stealth probing has the following secondary benefits:

- Encryption at the edge routers of a network infrastructure (even if selectively applied) (a) prevents the eavesdropping of unencrypted host-to-host communications, (b) prevents traffic-analysis attacks that host-to-host encryption does not prevent, for example, by hiding the source and destination addresses of data traffic, (c) precludes the adversary from impersonating the services of the receiving host, (d) renders misdirection attacks that divert traffic to adversarially controlled locations for eavesdropping and traffic analysis ineffective, and (e) enables ISPs to offer value-added services like VPNs.
- Stealth probing enforces fate sharing between data traffic and probes, which is broadly useful for troubleshooting network problems. For example, simple ICMP echo requests and replies may be treated differently from data packets either because of MTU size limits or packet filters that discard traffic based on the protocol or port numbers. Stealth probing avoids this problem by tunneling all traffic and matching the packet sizes of data and probe traffic (e.g., due to the padding step in active probing or the random packet sampling in passive probing).
- Tunnels are broadly useful for controlling the flow of traffic in an AS (e.g., for traffic engineering).

2.3 Mechanics

Stealth probing requires the endpoints of a path to share a secret and use this secret to create an *IPsec tunnel*. This section charts the workings of the IPsec protocol suite and the process that directs packets into tunnels.

IPsec protocol suite: IPsec provides end-to-end cryptographic protection at the IP layer. The communicating parties—the tunnel end-points—use the Internet Key

Exchange (IKE) protocol [7] to negotiate the establishment of a Security Association (SA). IKE relies on pre-shared secret keys or the public keys of an associated Public Key Infrastructure (PKI). In intradomain routing, key exchange can be assumed by a domain's authority; in interdomain routing, key exchange should not depend on a single central authority. Due to its end-to-end design, stealth probing does not depend on such authority.

Following the SA establishment, IP packets are protected using an Encapsulating Security Payload (ESP) module [9]. Using tunnel-mode ESP, the tunnel entry point adds an outer IP header to each packet, followed by the ESP header and trailer. ESP provides encryption using a standard encryption algorithm and ensures authenticity and integrity using a standard MAC. The tunnel exit point removes the outer IP header and restores the inner IP packet by inverting the encryption. Stealth probing, therefore, relies only on efficient symmetric cryptographic primitives. Thus, packet processing can proceed at the line speeds of core routers. Commercial routers increasingly offer such encryption capabilities.

Directing packets into tunnels: The management plane configures packet classifiers to specify which traffic should enter the tunnel, based on the five-tuples of source and destination address prefixes, port numbers, and protocol numbers. Tunnels are deployed across the network to match this specification (see Section 3). For protected packets, a longest-prefix-match table lookup will determine the tunnel exit point, based on a packet's destination address. A simple table lookup will then retrieve the associated encryption key needed to encapsulate the packet.

3 Deployment Scenarios

In this section, we present two deployment scenarios for stealth probing. First, we describe how an ISP can deploy stealth probing to secure its own infrastructure. Then, we discuss how a pair of edge networks can deploy stealth probing to secure the path through untrusted ASes in the Internet.

3.1 Intradomain Routing

Identifying tunnel endpoints: An ISP network typically has a *periphery* (i.e., edge routers that aggregate customer, transit, and peering traffic) and a *core* that interconnects the edge routers. The edge routers are an apt location to deploy stealth probing to leverage the benefits of an end-to-end design. First, core routers can be tunnel-agnostic and need only support simple destination-based forwarding and, second, processing requirements are distributed over a large number of edge routers. The man-

agement plane can configure five-tuples to identify the protected traffic, as discussed in Section 2.3.

The tunnel exit point corresponds to the next-hop attribute of the chosen BGP route for the destination prefix. A longest prefix match on a packet's destination address will determine the tunnel exit point (i.e., the egress router), and a simple table lookup returns the appropriate encryption key. In terms of scale, a large ISP network might have a few hundred edge routers, resulting in a few hundred keys and a few hundred tunnels per ingress point (i.e., one per egress router). Compared to the standard forwarding-table lookup that must be performed for each IP packet, the overhead of retrieving the keys is low; in fact, the forwarding table could store a pointer to the appropriate key for each prefix.

Byzantine tomography: In a network under attack, stealth probes detect the dysfunctional paths. Armed with this knowledge, the management plane can identify the compromised routers and recover from the attack. In the simplest case, the management plane can reconfigure or reboot the compromised routers, or reinstall the routers' operating system. Fine-grained detection of the compromised routers is useful to avoid the unnecessary downtime caused by false alarms. Byzantine tomography estimates the compromised routers by combining stealth probing output from multiple vantage points.

Byzantine tomography generalizes the notion of network tomography, which identifies the loss rates of network links using end-to-end probing traffic, by assuming that (the unknown) malicious routers may lie about their collected measurements. Byzantine tomography minimizes, over all possible faulty compositions, the number of faulty routers that explain the faulty paths observed in stealth probing. Algorithmically, this is an instance of the Minimum Hitting Set (MHS) problem: If S is the set of routers in the network and C is the collection of paths (subsets of S) that are faulty, a *hitting set* for C is a subset S' of S such that S' contains at least one element from each path in C . MHS can be solved using one of the algorithms presented in [3, 11].

The adversary's goal is to disorient the management plane into false detections. For example, the adversary can instruct the compromised routers to spuriously report certain paths as dysfunctional. If we require the routers to cryptographically sign their stealth-probing reports, a compromised router could not forge a bogus report for another router. As such, these reports could only identify paths that include the faulty router, making these reports accurate because the path does indeed contain a compromised router!

An adversary could also try to thwart the management system by selectively discarding packets traversing a small number of paths, making it difficult for Byzantine tomography to have fault reports from enough vantage

points to identify the compromised routers. However, in doing so, the adversary also confines the scope of attacks. The more selective the adversary is in dropping packets (to evade detection), the less extensive the damage of the attack. In addition, even if Byzantine tomography cannot uniquely identify the faulty routers, the network operators could easily take corrective action based on a *set* of suspected routers. For example, the operators could reconfigure the remaining routers to select paths that circumvent the suspected routers, or reboot each of the suspected routers.

3.2 Interdomain Routing

Securing interdomain routing is arguably harder than securing intradomain routing for two reasons. First, without a trusted central authority, key distribution is more challenging. Second, the compromised routers might reside in a remote AS outside the control of the communicating edge networks, making fault localization and fault recovery more challenging. An interdomain deployment can address these challenges through a small-scale key distribution (between selected edge networks) and coarse-grain rerouting (through techniques commonly used for intelligent route control).

Incremental deployability: ASes willing to deploy stealth probing over interdomain paths can engage in bilateral or small-scale multilateral agreements, and exchange pairwise keys either manually or by small-scale PKIs. ISPs have an incentive to join small groups, both to provide value-added services (such as multi-site VPNs) and to securely detect connectivity problems (to ensure higher availability for their services). Because stealth probing can be deployed across tunnel-agnostic legacy routers, early adopters will see an immediate benefit without requiring the participation of intermediate ASes. In fact, stealth probing enables the participating ASes to provide secure service, despite the presence of untrusted ASes in the rest of the Internet. The economic return to the early adopters can provide an incentive for other ASes to join these groups. As more ASes join these groups, scalable key distribution could be addressed through a larger PKI or a distributed trust model.

Circumventing the compromised routers: Although securely detecting routing failures is an important capability in its own right, the ability to bypass the affected routers is important as well. However, in interdomain routing, the communicating edge networks might not be able to identify the specific routers (or ASes) that have been compromised. Instead, the tunnel end points can adapt by directing the tunneled traffic on a different path, in the hope of circumventing the compromised routers, following techniques used in intelligent route control [1]. For example, consider two stub ASes, AS_1 and AS_2 ,

and assume that AS_1 is m_1 -multihomed and AS_2 is m_2 -multihomed. (For simplicity, also assume that each of AS_1 and AS_2 has a single border router.) AS_1 can choose among $m_1 \times m_2$ different BGP paths to forward traffic from AS_1 to AS_2 . Choosing any of the m_1 outgoing links is straightforward for AS_1 . Furthermore, any of the m_2 incoming links to AS_2 can be chosen as follows: AS_2 advertises a different primary prefix to each of its m_2 providers, and destination addresses from each of these prefixes are used to terminate m_2 tunnels between the border routers of AS_1 and AS_2 . AS_1 can, thus, direct traffic via any of the m_2 incoming links to AS_2 by choosing the remote tunnel end-point address accordingly. AS_2 selects the reverse path to AS_1 in the same manner. In this setting, stealth probing can detect which of the $m_1 \times m_2$ paths contain compromised routers, and the edge networks can switch to a working path.

4 Related Work

Encryption to make data and control traffic indistinguishable was first suggested by Perlman [16], who proposed hop-by-hop encryption between neighboring routers to hide beaconing traffic and prevent “man-in-the-middle” attacks on the topology-discovery process. The novelty of stealth probing is in applying this general idea to the paths between end-routers to identify data-plane problems in a secure fashion. Perlman also proposed recovery from routing attacks using multipath routing and disjoint paths. Stealth probing is well-suited for monitoring the quality of active paths in order to dynamically recompute the active path set.

The Fatih [13] secure data-plane monitor can adjust detection granularity from link-level to path-level for lower overhead. However, [13] does not propose a fault-localization mechanism to compensate for the reduced detection level, and the scheme also requires synchronized clocks. In our proposal, fault localization is attained using Byzantine tomography and we do not rely on clock synchronization.

Secure traceroute [15] is a link-level detection scheme that could conceivably be applied at the path level. Secure traceroute is based on secret identifiers embedded in packets that single out those packets as probes, which elicit responses for detecting reachability. However, this scheme may fail to detect attacks that target low-rate components of the aggregate traffic in a path or attacks that exploit the TCP mechanism. By encrypting traffic, stealth probing prevents those attacks. Secure traceroute could conceivably be extended into a hybrid scheme where the sender initiates link-level detection only after path-level probing suggests a problem. However, an adversary could easily thwart the on-demand link-level detection by limiting the duration of attacks; in addi-

tion, such a hybrid scheme would still require pairwise keys between the routers. In contrast, stealth probing, combined with Byzantine tomography, can pinpoint even short-lived attacks and does not require per-hop keys.

Other recent proposals, such as Listen [18] and Feedback-Based Routing [19], detect data-plane attacks by monitoring traffic at the TCP level. However, these techniques would falsely detect an unavailable path to a prefix as workable, if an adversary impersonates hosts in the monitored prefix.

5 Conclusion

In this paper, we presented stealth probing and Byzantine tomography as effective ways to protect against network-availability attacks without overburdening the data plane. We also showed how these techniques can be applied in the Internet's existing routing system, without changing the end hosts or the internal routers. In the future, we will explore "clean-slate" secure routing system designs. In particular, we will study whether more flexible path-selection schemes, such as source routing, are necessary, or whether coarse-grained path selection is sufficient for secure routing. We will also explore the many security benefits of encrypting the data traffic between edge networks, and study how to balance the trade-offs between host-based and network-based encryption for providing secure Internet services.

Acknowledgments

The authors would like to thank Constantinos Dovrolis, Nick Feamster, Karthik Lakshminarayanan, Barath Raghavan, Alex Snoeren, and the anonymous reviewers for their invaluable feedback. Ioannis Avramopoulos has been supported by a grant from the New Jersey Center for Wireless and Internet Security and a wireless testbed project (ORBIT) grant from the National Science Foundation. Jennifer Rexford was supported by Homeland Security Advanced Research Project Agency grant 1756303.

References

- [1] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman. A measurement-based analysis of multihoming. In *Proc. ACM SIGCOMM*, Aug. 2003.
- [2] I. Avramopoulos, H. Kobayashi, R. Wang, and A. Krishnamurthy. Highly secure and efficient routing. In *Proc. IEEE Infocom*, Mar. 2004.
- [3] I. Avramopoulos, A. Krishnamurthy, H. Kobayashi, and R. Wang. Nicephorus: Striking a balance between the recovery capability and the overhead of Byzantine detec-

- tion. Technical Report TR-710-04, Dept. of Computer Science, Princeton University, Oct. 2004.
- [4] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens. An on-demand secure routing protocol resilient to byzantine failures. In *Proc. ACM Workshop on Wireless Security*, Sep. 2002.
- [5] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, IETF, Jan. 1999.
- [6] N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Networking*, 9(3):280–292, Jun. 2001.
- [7] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409, IETF, Nov. 1998.
- [8] Y.-C. Hu, A. Perrig, and M. Sirbu. SPV: A secure path vector routing scheme for securing BGP. In *Proc. ACM SIGCOMM*, Sep. 2004.
- [9] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP). RFC 2406, IETF, Nov. 1998.
- [10] S. Kent, C. Lynn, and K. Seo. Secure Border Gateway Protocol (Secure-BGP). *IEEE Journal on Selected Areas in Communications*, 18(4):582–592, Apr. 2000.
- [11] R. Kompella, J. Yates, A. Greenberg, and A. Snoeren. IP fault localization via risk modeling. In *Proc. Symposium on Networked System Design and Implementation*, May 2005.
- [12] G. Mathur, V. Padmanabhan, and D. Simon. Securing routing in open networks using secure traceroute. Technical Report MSR-TR-2004-66, Microsoft Research, Jul. 2004.
- [13] A. Mizrak, Y.-C. Cheng, K. Marzullo, and S. Savage. Fatih: Detecting and isolating malicious routers. In *Proc. International Conference on Dependable Systems and Networks*, Jun. 2005.
- [14] S. Murphy, M. Badger, and B. Wellington. OSPF with digital signatures. RFC 2154, IETF, Jun. 1997.
- [15] V. Padmanabhan and D. Simon. Secure traceroute to detect faulty or malicious routing. In *Proc. ACM SIGCOMM HotNets Workshop*, Oct. 2002.
- [16] R. Perlman. *Network Layer Protocols with Byzantine Robustness*. PhD thesis, Massachusetts Institute of Technology, Aug. 1988.
- [17] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, S. Kent, and T. Strayer. Hash-based IP traceback. In *Proc. ACM SIGCOMM*, Aug. 2001.
- [18] L. Subramanian, V. Roth, I. Stoica, S. Shenker, and R. Katz. Listen and Whisper: Security mechanisms for BGP. In *Proc. Symposium on Networked System Design and Implementation*, Mar. 2004.
- [19] D. Zhu, M. Gritter, and D. Cheriton. Feedback based routing. In *Proc. ACM SIGCOMM HotNets Workshop*, Oct. 2002.

Service Placement in a Shared Wide-Area Platform

David Oppenheimer¹, Brent Chun², David Patterson³, Alex C. Snoeren¹, and Amin Vahdat¹

¹UC San Diego, ²Arched Rock Corporation, ³UC Berkeley
{doppenhe,snoeren,vahdat}@cs.ucsd.edu, bnc@theether.org, pattn@cs.berkeley.edu

ABSTRACT

Emerging federated computing environments offer attractive platforms to test and deploy global-scale distributed applications. When nodes in these platforms are time-shared among competing applications, available resources vary across nodes and over time. Thus, one open architectural question in such systems is how to map applications to available nodes—that is, how to discover and select resources. Using a six-month trace of PlanetLab resource utilization data and of resource demands from three long-running PlanetLab services, we quantitatively characterize resource availability and application usage behavior across nodes and over time, and investigate the potential to mitigate the application impact of resource variability through intelligent service placement and migration.

We find that usage of CPU and network resources is heavy and highly variable. We argue that this variability calls for intelligently mapping applications to available nodes. Further, we find that node placement decisions can become ill-suited after about 30 minutes, suggesting that some applications can benefit from migration at that timescale, and that placement and migration decisions can be safely based on data collected at roughly that timescale. We find that inter-node latency is stable and is a good predictor of available bandwidth; this observation argues for collecting latency data at relatively coarse timescales and bandwidth data at even coarser timescales, using the former to predict the latter between measurements. Finally, we find that although the utilization of a particular resource on a particular node is a good predictor of that node's utilization of that resource in the near future, there do not exist correlations to support predicting one resource's availability based on availability of other resources on the same node at the same time, on availability of the same resource on other nodes at the same site, or on time-series forecasts that assume a daily or weekly regression to the mean.

1. INTRODUCTION

Federated geographically-distributed computing platforms such as PlanetLab [3] and the Grid [7, 8] have recently become popular for evaluating and deploying network services and scientific computations. As the size, reach, and user population of such infrastructures grow, resource discovery and resource selection become increas-

ingly important. Although a number of resource discovery and allocation services have been built [1, 11, 15, 22, 28, 33], there is little data on the utilization of the distributed computing platforms they target. Yet the design and efficacy of such services depends on the characteristics of the target platform. For example, if resources are typically plentiful, then there is less need for sophisticated allocation mechanisms. Similarly, if resource availability and demands are predictable and stable, there is little need for aggressive monitoring.

To inform the design and implementation of emerging resource discovery and allocation systems, we examine the usage characteristics of PlanetLab, a federated, time-shared platform for “developing, deploying, and accessing” wide-area distributed applications [3]. In particular, we investigate variability of available host resources across nodes and over time, how that variability interacts with resource demand of several popular long-running services, and how careful application placement and migration might reduce the impact of this variability. We also investigate the feasibility of using stale or predicted measurements to reduce overhead in a system that automates service placement and migration.

Our study analyzes a six-month trace of node, network, and application-level measurements. In addition to presenting a detailed characterization of application resource demand and free and committed node resources over this time period, we analyze this trace to address the following questions: (i) Could informed service placement—that is, using live platform utilization data to choose where to deploy an application—outperform a random placement? (ii) Could migration—that is, moving deployed application instances to different nodes in response to changes in resource availability—potentially benefit some applications? (iii) Could we reduce the overhead of a service placement service by using stale or predicted data to make placement and migration decisions? We find:

- CPU and network resource usage are heavy and highly variable, suggesting that shared infrastructures such as PlanetLab would benefit from a resource allocation infrastructure. Moreover, available resources across nodes and resource demands across instances of an application both vary widely. This suggests that even in the absence of a resource allocation system, some applications could benefit from intel-

ligently mapping application instances to available nodes.

- Node placement decisions can become ill-suited after about 30 minutes, suggesting that a resource discovery system should not only be able to deploy applications intelligently, but should also support migrating performance-sensitive applications whose migration cost is acceptable.
- Stale data, and certain types of predicted data, can be used effectively to reduce measurement overhead. For example, using resource availability and utilization data up to 30 minutes old to make migration decisions still enables our studied applications' resource needs to be met more frequently than not migrating at all; this suggests that a migration service for this workload need not support a high measurement update rate. In addition, we find that inter-node latency is both stable and a good predictor of available bandwidth; this observation argues for collecting latency data at relatively coarse timescales and bandwidth data at even coarser timescales, using the former to predict the latter between measurements.
- Significant variability in usage patterns across applications, combined with heavy sharing of nodes, precludes significant correlation between the availability of different resources on the same node or at the same site. For example, CPU availability does not correspond to memory or network availability on a particular node, or to CPU availability on other nodes at the same site. Hence, it is not possible to make accurate predictions based on correlations within a node or a site. Furthermore, because PlanetLab's user base is globally distributed and applications are deployed across a globally distributed set of nodes, we note an absence of the cyclic usage pattern typical of Internet services with geographically colocated user populations. As a result, it is not possible to make accurate resource availability or utilization predictions for this platform based on time-series forecasts that assume a daily or weekly regression to the mean.

The remainder of this paper is organized as follows. Section 2 describes our data sources and methodology. Section 3 surveys platform, node, and network resource utilization behavior; addresses the usefulness of informed service placement; and describes resource demand models for three long-running PlanetLab services—CoDeeN [26], Coral [10], and OpenDHT [20]—that we use there and in subsequent sections. Section 4 investigates the potential benefits of periodically migrating service instances. Section 5 analyzes the feasibility of making placement and

migration decisions using stale or predicted values. Section 6 discusses related work, and Section 7 concludes.

2. DATA SOURCES AND METHODOLOGY

We begin by describing our measurement data sources and PlanetLab, our target computing infrastructure. We then describe our study's methodology and assumptions.

2.1 System environment

PlanetLab is a large-scale federated computing platform. It consists of over 500 nodes belonging to more than 150 organizations at over 250 sites in more than 30 countries. In general, approximately two-thirds of these nodes are functioning at any one time. PlanetLab currently performs no coordinated global scheduling, and users may deploy applications on any set of nodes at any time.

All PlanetLab nodes run identical versions of Linux on x86 CPUs. Applications on PlanetLab run in *slices*. A slice is a set of allocated resources distributed across platform nodes. From the perspective of an application deployer, a slice is roughly equivalent to a user in a traditional Unix system, but with additional resource isolation and virtualization [3], and with privileges on many nodes. The most common slice usage pattern is for an application deployer to run a single distributed application in a slice. This one-to-one correspondence between slices and applications is true for the applications we characterize in Section 3.2, and for many other slices as well. The set of allocated resources on a single node is referred to as a *sliver*. When we discuss “migrating a sliver,” we mean migrating the process(es) running on behalf of one slice on one node, to another node. We study approximately six months of PlanetLab node and network resource utilization data, from August 12, 2004 to January 31, 2005, collected from the following data sources. All sources periodically archive their measurements to a central node.

CoTop [17] collects data every 5 minutes on each node. It collects node-level information about 1-, 5-, and 15-minute load average; free memory; and free swap space. Additionally, for each sliver, *CoTop* collects a number of resource usage statistics, including average send and receive network bandwidth over the past 1 and 15 minutes, and memory and CPU utilization. *All-pairs pings* [24] measures the latency between all pairs of PlanetLab nodes every 15 minutes using the Unix “ping” command. *iPerf* [5] measures the available bandwidth between every pair of PlanetLab nodes once to twice a week, using a bulk TCP transfer.

In this study, we focus on CPU utilization and network bandwidth utilization. With respect to memory, we observed that almost all PlanetLab nodes operated with their physical memory essentially fully committed on a contin-

uous basis, but with a very high fraction of their 1 GB of swap space free. In contrast, CPU and network utilization were highly variable.

In our simulations, we use CoTop’s report of per-sliver network utilization as a measure of the sliver’s network bandwidth demand, and we assume a per-node bandwidth capacity of 10 Mb/s, which was the default bandwidth limit on PlanetLab nodes during the measurement period. We use CoTop’s report of per-sliver %CPU utilization as a proxy for the sliver’s CPU demand. We use $\frac{1}{load+1} * 100\%$ to approximate the %CPU that would be available to a new application instance deployed on a node. Thus, for example, if we wish to deploy a sliver that needs 25% of the CPU cycles of a PlanetLab node and 1 Mb/s of bandwidth, we say that it will receive “sufficient” resources on any node with $load \leq 3$ and on which competing applications are using at most a total of 9 Mb/s of bandwidth.

Note that this methodology of using observed resource utilization as a surrogate for demand tends to under-estimate true demand, due to resource competition. For example, TCP congestion control may limit a sliver’s communication rate when a network link the sliver is using cannot satisfy the aggregate demand on that link. Likewise, when aggregate CPU demand on a node exceeds the CPU’s capabilities, the node’s process scheduler will limit each sliver’s CPU usage to its “fair share.” Obtaining true demand measurements would require running each application in isolation on a cluster, subjecting it to the workload it received during the time period we studied. Note also that the resource demand of a particular sliver of an isolated application may change once that application is subjected to resource competition, even if the workload remains the same, if that competition changes how the application distributes work among its slivers. (Only one of the applications we studied took node load into account when assigning work; see Section 3.2.)

Finally, recent work has shown that using $\frac{1}{load+1} * 100\%$ to estimate the %CPU available to a new process underestimates actual available %CPU, as measured by an application running just a spin-loop, by about 10 percentage points most of the time [23]. The difference is due to PlanetLab’s proportional share node CPU scheduling policy, which ensures that all processes of a particular sliver together consume no more than $\frac{1}{m}$ of the CPU, where m is the number of slivers demanding the CPU at that time. Spin-loop CPU availability measurements are not available for the time period in our study.

2.2 Methodology

We take a two-pronged approach to answering the questions posed in Section 1. First we examine node-level measurement data about individual resources, independent of any application model. We then compose this data

about available node resources with simple models of application resource demand derived from popular PlanetLab services. This second step allows us to evaluate how space-and-time-varying sliver resource demand interacts with space-and-time-varying free node resources to determine the potential benefits of informed sliver placement and migration, as well as the impact of optimizations to a migration service.

We ask our questions from the perspective of a single application at a time; that is, we assume all other applications behave as they did in the trace. Thus, our approach combines *measurement*, *modeling*, and *simulation*, but the simulation does not consider how other users might react to one user’s placement and migration decisions. We leave to future work developing a multi-user model that might allow us to answer the question “if multiple PlanetLab users make placement and migration decisions according to a particular policy, how will the system evolve over time?”

3. DEPLOYMENT-TIME PLACEMENT

If the number of individuals wishing to deploy applications in a shared distributed platform is small, or the number of nodes each user needs is small, a platform architect might consider space-sharing nodes rather than time-sharing. Giving users their own dedicated set of nodes eliminates node resource competition and thereby removes the primary motivation of informed application placement—finding lightly utilized nodes and avoiding overutilized ones. Therefore, we first ask whether such a space-sharing scheme is practical, by investigating what portion of global resources application deployers typically wish to use.

Figure 1 shows the number of PlanetLab nodes used by each slice active during our measurement period. We say a slice is “using” a node if the slice has at least one task in the node’s process table (i.e., the slice has processes on that node, but they may be running or sleeping at the time the measurement is taken). More than half of PlanetLab slices run on at least 50 nodes, with the top 20% operating on 250 or more nodes. This is not to say, of course, that all of those applications “need” that many nodes to function; for example, if a platform like PlanetLab charged users per node they use, we expect that this distribution would shift towards somewhat lower numbers of nodes per slice. However, we expect that a desire to maximize realism in experimental evaluation, to stress-test application scalability, to share load among nodes for improved performance, and/or to maximize location diversity, will motivate wide-scale distribution even if such distribution comes at an economic cost. Therefore, based on the data we collected, we conclude that PlanetLab applications *must* time-share nodes, because it is unlikely that a platform like PlanetLab would

ever grow to a size that could accommodate even a few applications each “owning” hundreds of nodes on an on-going basis, not to mention allowing those large-scale applications to coexist with multiple smaller-scale experiments. And it is this best-effort time-sharing of node resources that leads to variable per-node resource availability over time on PlanetLab.

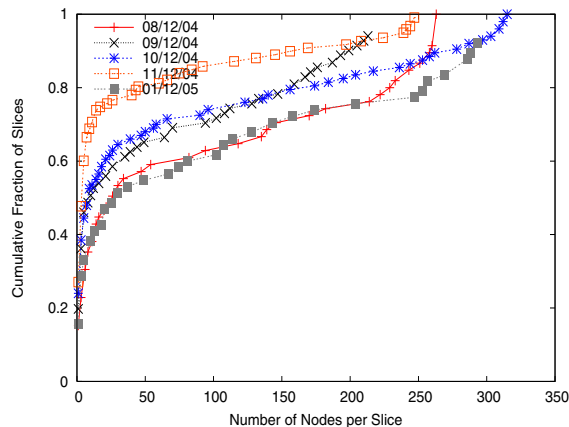


Figure 1: Cumulative distribution of nodes per slice averaged over the first day of each month.

At the same time, more than 70% of slices run on less than half of the nodes, suggesting significant flexibility in mapping slices to nodes. Of course, even applications that wish to run on all platform nodes may benefit from intelligent resource selection, by mapping their most resource-intensive slivers to the least utilized nodes and vice-versa.

Having argued for the necessity of time-sharing nodes, the remainder of this section quantifies resource variability across nodes and investigates the potential for real applications to exploit that heterogeneity by making informed node selection decisions at deployment time. We observe significant variability of available resources across nodes, and we find that in simulation, a simple load-sensitive sliver placement algorithm outperforms a random placement algorithm. These observations suggest that some applications are likely to benefit from informed placement at deployment time.

3.1 Resource heterogeneity across nodes

Table 1 shows that static per-node PlanetLab node characteristics are fairly homogeneous across nodes. However, dynamic resource demands are heavy and vary over both space and time. To quantify such variation, we consider the 5-minute load average on all PlanetLab nodes at three instants in our 6-month trace. These instants correspond to an instant of low overall platform utilization, an instant of typical platform utilization, and an instant of high platform utilization. Figure 2 shows a CDF of the

attribute	mean	std. dev.	median	10th %ile	90th %ile
# of CPUs	1.0	0.0	1.0	1.0	1.0
cpu speed (MHz)	2185	642.9	2394	1263	3066
total disk (GB)	98	48	78	69	156
total mem (MB)	1184	364	1036	1028	2076
total swap (GB)	1.0	0.0	1.0	1.0	1.0

Table 1: Static node measurements on Feb 18, 2005. Similar results were found during all other time periods as well. PlanetLab nodes are also geographically diverse, located in over 30 countries.

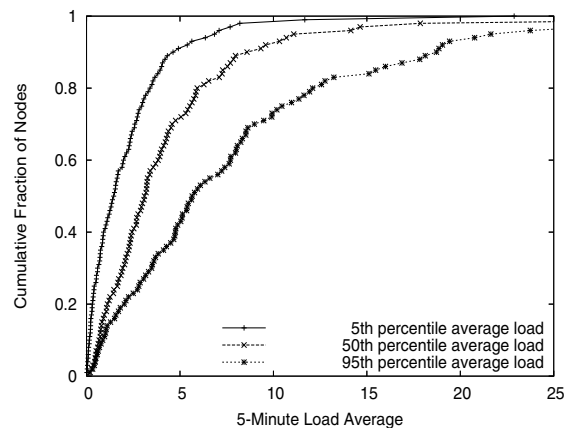


Figure 2: CDF of node loads at three representative moments in the trace: when load averaged across all nodes was “typical” (median), “low” (5th percentile), and “high” (95th percentile).

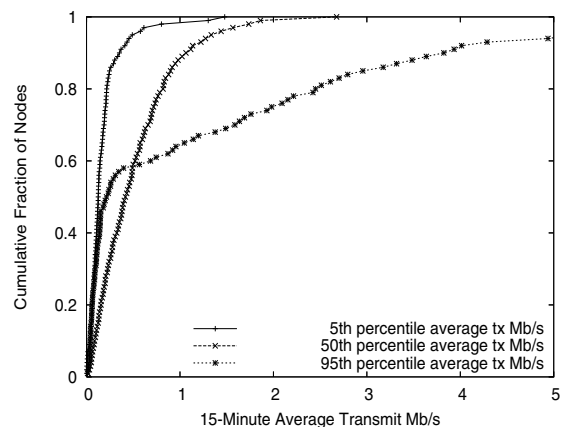


Figure 3: CDF of node 15-minute transmit bandwidths at three representative moments in the trace: when node transmit bandwidth averaged across all nodes was “typical” (median), “low” (5th percentile), and “high” (95th percentile). We found similar results for 15-minute receive bandwidth.

5-minute load average across all nodes at three separate moments in time: the time when the load averaged across all nodes was the 5th percentile of such averages over the entire trace (low platform utilization), the time when the load averaged across all nodes was the median of such averages over the entire trace (typical platform utilization), and the time when the load averaged across all nodes was the 95th percentile of such averages over the entire trace (high platform utilization). Analogously, Figure 3 shows a CDF of the 15-minute average transmit bandwidth across all nodes at three moments: low, typical, and high utilization of platform-wide transmit bandwidth. Results for receive bandwidth were similar.

We see that load varies substantially across nodes independent of overall system utilization, while network bandwidth utilization varies substantially across nodes primarily when the platform as a whole is using significantly higher-than-average aggregate bandwidth. This suggests that load is always a key metric when making application placement decisions, whereas available per-node network bandwidth is most important during periods of peak platform bandwidth utilization. Note that periods of low platform bandwidth utilization may be due to low aggregate application demand or due to inability of the network to satisfy demand.

Not only per-node attributes, but also inter-node characteristics, vary substantially across nodes. Figure 4 shows significant diversity in latency and available bandwidth among pairs of PlanetLab nodes.

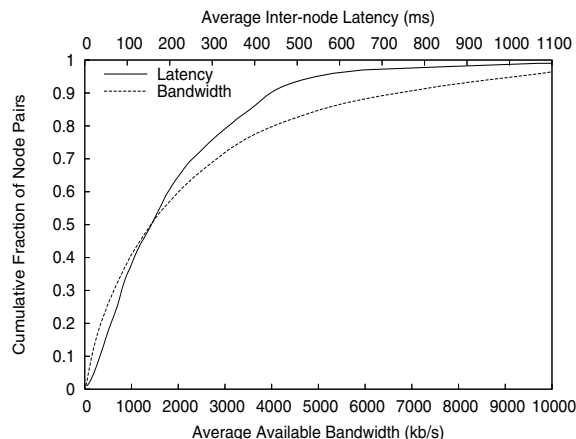


Figure 4: CDF of node-pair latencies and available bandwidths over the entire trace.

Furthermore, we find that PlanetLab nodes exhibit a wide range of MTTFs and MTTRs. Figure 5 shows MTTF and MTTR based on “uptime” measurements recorded by nodes. We declare that a node has failed when its uptime decreases between two measurement intervals. We declare the node to have failed at the time we received the last measurement report with a monotonically increasing

uptime, and to have recovered at $t_{report} - \Delta t_{up}$ where t_{report} is the time we receive the measurement report indicating an uptime less than the previous measurement report, and Δt_{up} is the uptime measurement in that report. Computing MTTF and MTTR based on “all-pairs pings” data, with a node declared dead if no other node can reach it, yields similar results. Compared to the uptime-based approach, MTTF computed using pings is slightly lower, reflecting situations where a node becomes disconnected from the network but has not crashed, while MTTR computed using pings is slightly higher, reflecting situations in which some time passes between a node’s rebooting and the restarting of network services.

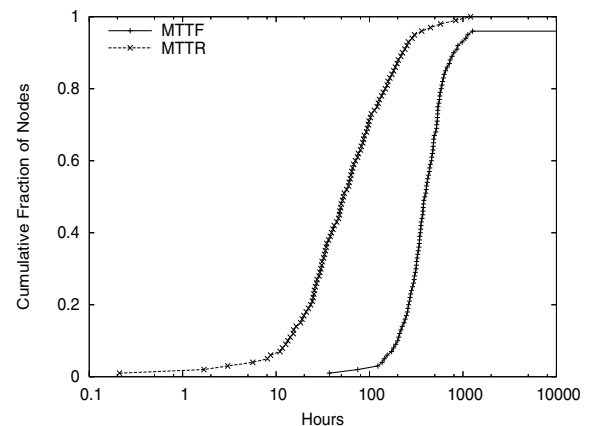


Figure 5: CDF of MTTF and MTTR based on “uptime.” These curves show median availability of 88% and 95th percentile availability of 99.7%.

These observations of substantial heterogeneity across node and node-pair attributes suggest that there is much available resource diversity for a service placement mechanism to exploit.

3.2 Nodes meeting application requirements

Having observed significant heterogeneity across nodes, we now examine how that heterogeneity combines with the resource requirements of real applications to dictate the potential usefulness of informed application placement. In order to do this, we first must develop models of these applications’ resource demands. In particular, we study resource usage by three long-running PlanetLab services.

1. **CoDeeN** [26] is a prototype Content Distribution Network. It has been operating since June 2003 and currently receives about 10 million hits per day from about 30,000 users. Over the course of the trace, CoDeeN slivers ran on 384 unique nodes.
2. **Coral** [10] is another Content Distribution Network. It has been operating since August 2004 and cur-

rently receives 10-20 million hits per day from several hundred thousand users. Over the course of the trace, Coral slivers ran on 337 unique nodes.

3. **OpenDHT** [20] is a publicly accessible distributed hash table. OpenDHT has been operating since mid-2004. Over the course of the trace, OpenDHT slivers ran on 406 unique nodes.

We chose these services because they are continuously-running, utilize a large number of PlanetLab nodes, and serve individuals outside the computer science research community. They are therefore representative of “production” services that aim to maximize user-perceived performance while coexisting with other applications in a shared distributed infrastructure. They are also, to our knowledge, the longest-running non-trivial PlanetLab services. Note that during the time period of our measurement trace, these applications underwent code changes that affect their resource consumption. We do not distinguish changes in resource consumption due to code upgrades from changes in resource consumption due to workload changes or other factors, as the reason for a change in application resource demand is not important in addressing the questions we are investigating, only the effect of the change on application resource demand.

We study these applications’ resource utilization over time and across nodes to develop a loose model of the applications’ resource needs. We can then combine that model with node-level resource data to evaluate the quality of different mappings of slivers (application instances) to nodes. Such an evaluation is essential to judging the potential benefit of service placement and migration.

Figures 6, 7, and 8 show CPU demand and outgoing network bandwidth for these applications over time, using a log-scale Y-axis. Due to space constraints we show only a subset of the six possible graphs, but all graphs displayed similar characteristics. Each graph shows three curves, corresponding to the 5th percentile, median, and 95th percentile resource demand across all slivers in the slice at each timestep. (For example, the sliver whose resource demand is the 95th percentile at time t may or may not be the same sliver whose resource demand is the 95th percentile at time $t + n$.)

We see that a service’s resource demands vary significantly across slivers at any given point in time. For example, all three graphs show resource demand varying by at least an order of magnitude from the median sliver to the 95th percentile sliver. These results suggest that not only available host resources, but also variable per-sliver resource demand, must be considered when making placement decisions.

One interpretation of widely varying per-sliver demands is that applications are performing internal load balancing, i.e., assigning more work to slivers running on nodes

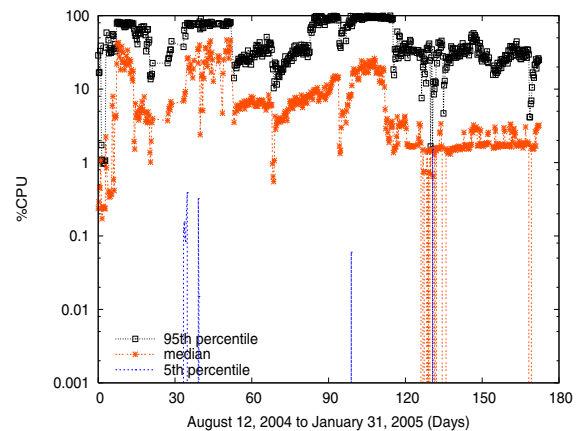


Figure 6: CPU resource demand for OpenDHT, using a log-scale Y-axis.

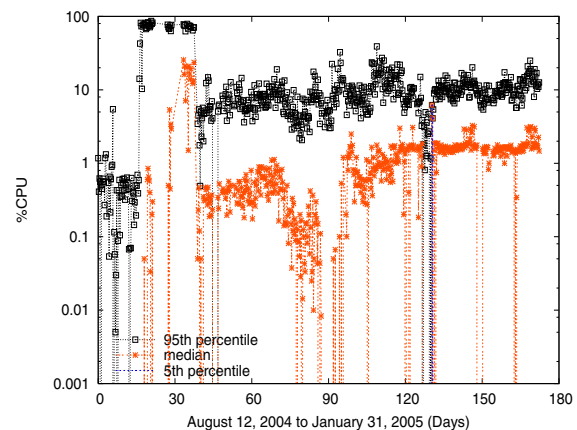


Figure 7: CPU resource demand for Coral, using a log-scale Y-axis.

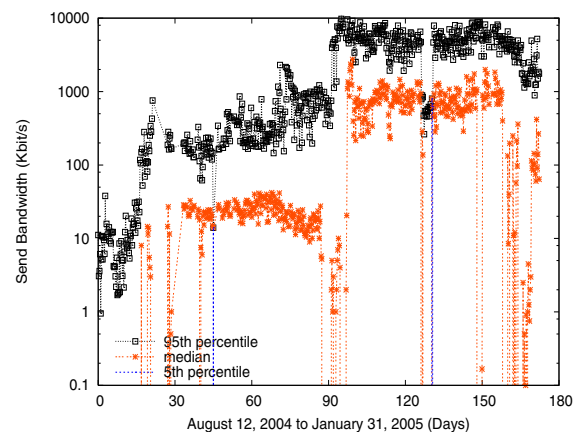


Figure 8: Transmit bandwidth demand for Coral, using a log-scale Y-axis.

with ample free resources. However, these three applications principally use a hash of a request's contents to map requests to nodes, suggesting that variability in per-node demands is primarily attributable to the application's external workload and overall structure, not internal load balancing. For instance, OpenDHT performs no internal load balancing; Coral's Distributed Sloppy Hash Table balances requests to the same key ID but this balancing does not take into account available node resources; and CoDeeN explicitly takes node load and reliability into account when choosing a reverse proxy from a set of candidate proxies.

Given this understanding of the resource demands of three large-scale applications, we can now consider the potential benefits of informed service placement. To answer this question, we simulate deploying a *new* instance of one of our three modeled applications across Planet-Lab. We assume that the new instance's slivers will have resource demands identical to those of the original instance. We further assume that the new instance will use the same set of nodes as the original, to allow for the possibility that the application deployer intentionally avoided certain nodes for policy reasons or because of known availability problems. Within these constraints, we allow for any possible one-to-one mapping of slivers to nodes. Thus, our goal is to determine how well a particular service placement algorithm will meet the requirements of an application, knowing both the application's resource demands and available resources of the target infrastructure.

We simulate two allocation algorithms. The *random* algorithm maps slivers to nodes randomly. The *load-sensitive* algorithm deploys heavily CPU-consuming slivers onto lightly loaded nodes and vice-versa. In both cases, each sliver's resource consumption is taken from the sliver resource consumption measurements at the corresponding timestep in the trace, and each node's amount of free resources is calculated by applying the formulas discussed in Section 2 to the node's load and bandwidth usage indicated at that timestep in the trace. We then calculate, for each timestep, the fraction of slivers whose assigned nodes have "sufficient" free CPU and network resources for the sliver, as defined in Section 2. If our *load-sensitive* informed service placement policy is useful, then it will increase, relative to random placement, the fraction of slivers whose resource needs are met by the nodes onto which they are deployed. Of course, if a node does not meet a sliver's resource requirements, that sliver will still function from a practical standpoint, but its performance will be impaired.

Figures 9 and 10 show the fraction of nodes that meet application CPU and network resource needs, treating each timestep in the trace as a separate deployment. As Figures 2 and 3 imply, CPU was a more significant bottle-

neck than node access link bandwidth. We see that the load-sensitive placement scheme outperforms the random placement scheme, increasing the number of slivers running on nodes that meet their resource requirements by as much as 95% in the case of OpenDHT and as much as 55% in the case of Coral (and CoDeeN, the graph of which is omitted due to space constraints). This data argues that there is potentially significant performance improvement to be gained by using informed service placement based on matching sliver resource demand to nodes with sufficient available resources, as compared to a random assignment. A comprehensive investigation of what application characteristics make informed node selection more beneficial or less beneficial for one application compared to another is left to future work.

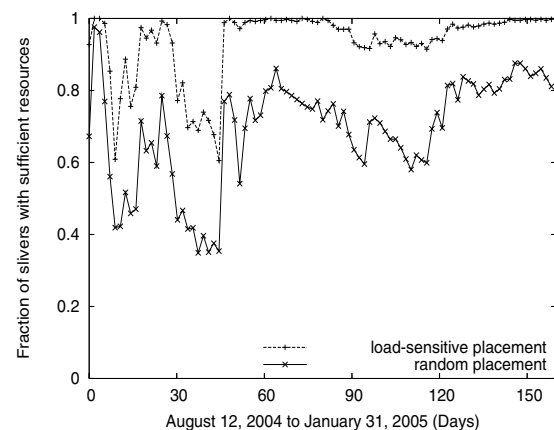


Figure 9: Fraction of OpenDHT slivers whose resource requirements are met by the node onto which they are deployed, vs. deployment time.

4. SLIVER MIGRATION

We have seen that applications can potentially benefit from intelligently mapping their slivers to nodes based on sliver resource demand and available node resources. Our next question is whether migrating slivers could improve overall application performance—that is, whether, and how often, to periodically recompute the mapping. While process migration has historically proven difficult, many distributed applications are designed to gracefully handle node failure and recovery; for such applications, migration requires simply killing an application instance on one node and restarting it on another node. Furthermore, emerging virtual machine technology may enable low-overhead migration of a sliver without resorting to exit/restart. Assuming the ability to migrate slivers, we consider the potential benefits of doing so in this section. Of course, migration is feasible only for services that do not need to “pin” particular slivers to particular nodes. For example, sliver location is not “pinned” in services

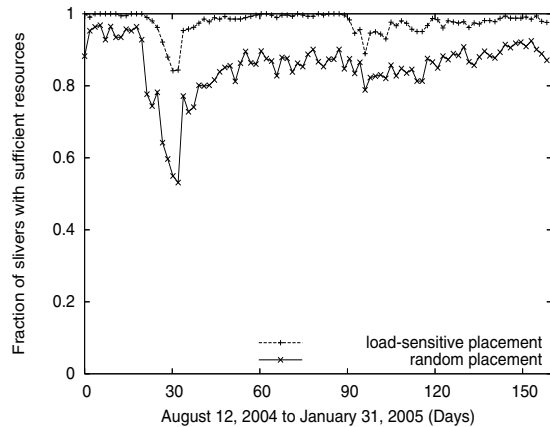


Figure 10: Fraction of Coral slivers whose resource requirements are met by the node onto which they are deployed, vs. deployment time. We note that Coral’s relatively low per-sliver CPU resource demands result in a larger fraction of its slivers’ resource demands being met relative to OpenDHT.

that map data to nodes pseudo-randomly by hashing the contents of requests, as is the case (modulo minor implementation details) for the three typical PlanetLab applications we have studied. We comment on the applicability of these results to additional application classes in Section 7.

Before considering the potential benefits of migration, we must first determine the typical lifetime of individual slivers. If most slivers are short-lived, then a complex migration infrastructure is unnecessary since per-node resource availability and per-sliver resource demand are unlikely to change significantly over very short time scales. In that case making sliver-to-node mapping decisions only when slivers are instantiated, i.e., when the application is initially deployed and when an existing sliver dies and must be re-started, should suffice. Figure 11 shows the average sliver lifetime for each slice in our trace. We see that slivers are generally long-lived: 75% of slices have average sliver lifetimes of at least 6 hours, 50% of slices have average sliver lifetimes of at least two days, and 25% of slices have average sliver lifetimes of at least one week. As before, we say that a sliver is “alive” on a node if it appears in the process table for that node.

To investigate the potential usefulness of migration, we next examine the rate of change of available node resources. If per-node resource availability varies rapidly relative to our measurements of sliver lifetimes, we can hypothesize that sliver migration may be beneficial.

4.1 Node resource variability over time

To assess the variability of per-node available resources over time, we ask what fraction of nodes that meet a par-

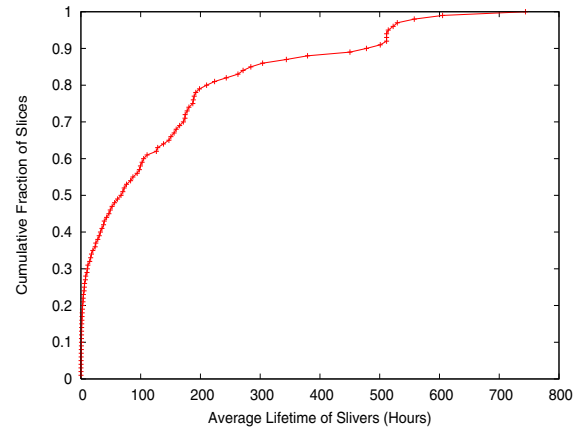


Figure 11: CDF of fraction of slices vs. average sliver lifetime for that slice. A sliver may die due to software failure or node crash; when the sliver comes back up, we count it as a new sliver.

ticular resource requirement at time T continue to meet that requirements for all time intervals between T and $T + x$, for various values of x and all times T in our trace. If the fraction is large, then most slivers initially deployed to nodes meeting the requirement at time T will find themselves continuously executing on nodes that meet the requirement until time $T + x$. Conversely, if the fraction is small, then most slivers will find the resource requirement violated at some time before $T + x$, suggesting that they may benefit from migration at a time granularity on the order of x .

Figures 12 and 13 show the fraction of nodes that meet a particular resource requirement at time T that continue to meet the requirement for all time intervals between T and $T + x$, for various values of x , averaged over all starting times T in our trace. The fraction of nodes that continually meet initial requirements declines rapidly with increasing intervals x , and the rate of decline increases with the stringency of the requirement. Most importantly, we see that the fraction of nodes continually meeting typical resource requirements remains relatively high (80% or greater) up to about 30 minutes post-deployment for load and up to about 60 minutes post-deployment for network traffic. This result suggests that if sliver resource requirements remain relatively constant over time, then it is unnecessary to migrate more often than every 30 to 60 minutes.

4.2 Sliver suitability to nodes over time

The suitability of a particular node to host a particular sliver depends not only on the resources available on that node, but also on the resource demands of that sliver over time. We therefore perform an analysis similar to that in the previous section, but accounting for

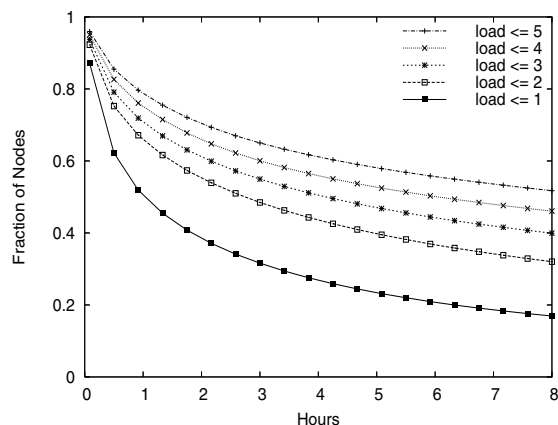


Figure 12: Fraction of nodes continuously meeting various load constraints for various durations after initially meeting the constraint. The fraction is 100% at $x = 0$ because we consider only nodes that initially meet the constraint.

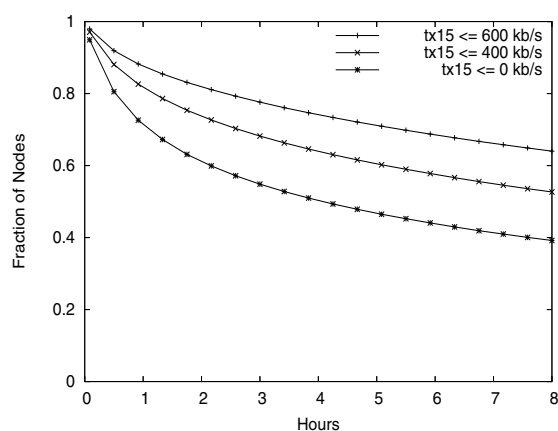


Figure 13: Fraction of nodes continuously meeting various constraints on network transmit bandwidth from competing applications for various durations after initially meeting the constraint. The fraction is 100% at $x = 0$ because we consider only nodes that initially meet the constraint. Similar results were found for receive bandwidth.

both available node resources and application resource demand. Here we are interested not in the stability of available resources on individual nodes, but rather in the stability of the fraction of slivers whose resource requirements are met after deployment. It is the rate of decline of this fraction that dictates an appropriate migration interval for the application—very rapid decline will require prohibitively frequent migration, while very slow decline means migration will add little to a simple policy of intelligent initial sliver placement and re-deployment upon failure.

Thus, we ask what fraction of nodes onto which slivers are deployed at time T meet the requirements of their sliver at time $T + x$, for various values of x . For each $T + x$ value, we average this measure over every possible deployment time T in our trace. A large fraction means that most slivers will be running on satisfactory hosts at the corresponding time. As in Section 3.2, we say a node meets a sliver’s requirements if the node has enough free CPU and network bandwidth resources to support a new sliver assigned from the set of sliver resource demands found at that timestep in the trace, according to the load-sensitive or random placement policy.

Figures 14 and 15 show the fraction of slivers whose resource requirements were met at the time indicated on the X-axis, under both the random and load-sensitive schemes for initially mapping slivers to nodes at time $X = 0$. Note that the random placement line is simply a horizontal line at the value corresponding to average across all time intervals from Figure 9 in the case of OpenDHT and Figure 10 in the case of Coral.

We make two primary observations from these graphs. First, the quality of the initially load-sensitive assignment degrades over time as node resources and sliver demands become increasingly mismatched. This argues for periodic migration to re-match sliver needs and available host resources. Second, the benefit of load-sensitive placement over random placement—the distance between the load-sensitive and random lines—erodes over time for the same reason, but persists. This persistence suggests that informed initial placement can be useful even in the absence of migration.

Choosing a desirable migration period requires balancing the cost of migrating a particular application’s slivers against the rate at which the application mapping’s quality declines. For example, in OpenDHT, migration is essentially “free” since data is stored redundantly—an OpenDHT instance can be killed on one node and re-instantiated on another node (and told to “own” the same DHT key range as before) without causing the service to lose any data. Coral and CoDeeN can also be migrated at low cost, as they are “soft state” services, caching web sites hosted externally to their service. An initially load-sensitive sliver mapping for OpenDHT has declined to close to its asymptotic value within 30 minutes, arguing for migrating poorly-matched slivers at that timescale or less. If migration takes place every 30 minutes, then the quality of the match will, on average, traverse the curve from $t = 0$ to $t = 30$ every 30 minutes, returning to $t = 0$ after each migration. Coral and CoDeeN placement quality declines somewhat more quickly than OpenDHT, but migrating poorly matched slivers of these services on the order of every 30 minutes is unlikely to cause harm and will allow the system to maintain a somewhat better mapping than would be achieved with a less aggressive

migration interval.

A comprehensive investigation of what application characteristics make migration more beneficial or less beneficial for one application compared to another is left to future work, as is emulation-based verification of our results (i.e., implementing informed resource selection and migration in real PlanetLab applications, and measuring user-perceived performance with and without those techniques under repeatable system conditions). Our focus in this paper is a simulation-based analysis of whether designers of future resource selection systems should consider including informed placement and migration capabilities, by showing that those techniques are potentially beneficial for several important applications on a popular existing platform.

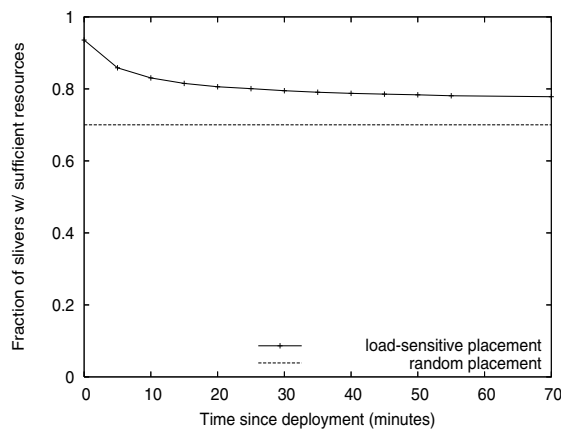


Figure 14: Fraction of OpenDHT slivers hosted on nodes that meet the sliver's requirements at the time that is indicated on the X-axis.

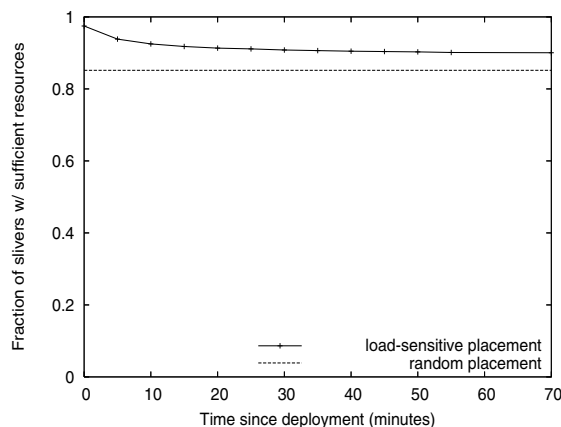


Figure 15: Fraction of Coral slivers hosted on nodes that meet the sliver's requirements at the time that is indicated on the X-axis. CoDeeN showed similar results.

5. DESIGN OPTIMIZATIONS

Our preceding experiments have assumed that resource availability data is collected from every node every 5 minutes, the minimum time granularity of our trace. In a large-scale system, it may be undesirable to collect data about every resource attribute from all nodes that frequently. Thus, we investigate two optimizations that a service placement and migration service might use to reduce measurement overhead. First, the system might simply collect node measurement data less frequently, accepting the tradeoff of reduced accuracy. Second, it might use statistical techniques to predict resource values for one resource based on measurements it has collected of other resource values on the same node, resource values on other nodes, or historical resource values.

5.1 Reducing measurement frequency

In this section, we examine the impact of reducing measurement frequency, first by quantifying the relative measurement error resulting from relaxed data measurement intervals, and then by studying the impact that stale data has on migration decisions for our modeled applications.

5.1.1 Impact on measurement accuracy

Figures 16, 17, and 18 show the accuracy impact of relaxed measurement intervals for load, network transmit bandwidth, and inter-node latency. For each of several update intervals longer than 5 minutes, we show the fraction of nodes (or, in the case of latency, node pairs) that incur various average absolute value errors over the course of the trace, compared to updates every 5 minutes (15 minutes for latency).

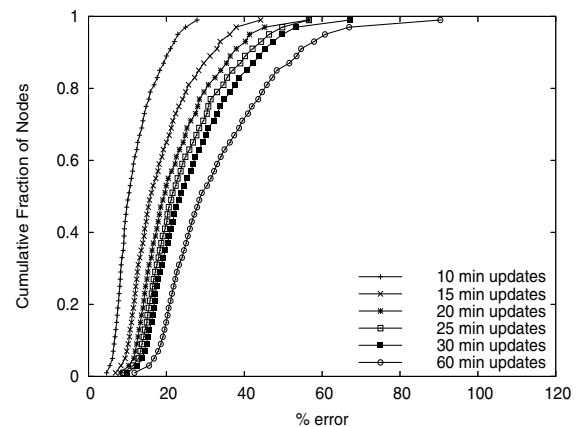


Figure 16: Mean error in 5-minute load average compared to 5-minute updates.

We observe several trends from these graphs. First, latency is more stable than load or network transmit bandwidth. For example, if a maximum error of 20% is tolerable, then moving from 15-minute measurements to hourly

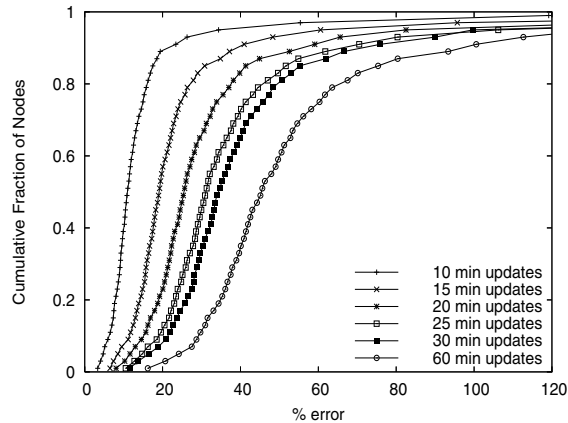


Figure 17: Mean error in 15-minute average network transmit bandwidth compared to 5-minute updates. Similar results were found for receive bandwidth.

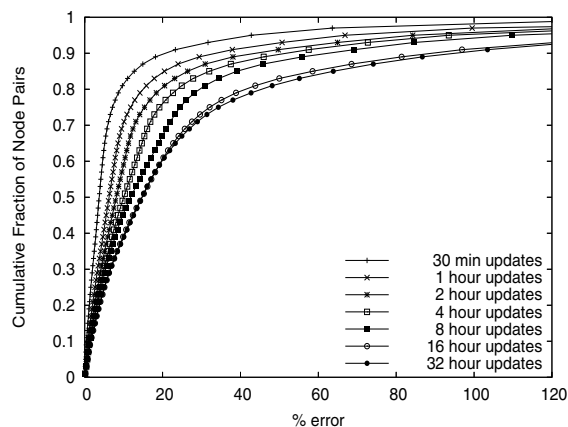


Figure 18: Mean error in inter-node latency compared to 15-minute updates.

measurements for latency will push about 12% of node pairs out of the 20% accuracy range, but moving from 15-minute measurements to hourly measurements for load and network bandwidth usage will push about 50% and 55% of nodes, respectively, out of the 20% accuracy range. Second, network latency, and to a larger extent network bandwidth usage, show longer tails than does load. For example, with a 30-minute update interval, only 3% of nodes show more than 50% error in load, but over 20% of nodes show more than 50% error in network bandwidth usage. This suggests that bursty network behavior is more common than bursty CPU load.

From these observations, we conclude that for most nodes, load, network traffic, and latency data can be collected at relaxed update intervals, e.g., every 30 minutes, without incurring significant error. Of course, the exact amount of tolerable error depends on how the measurement data is being used. Further, we see that a small

number of nodes show significant burstiness with respect to network behavior, suggesting that a service placement infrastructure could maximize accuracy while minimizing overhead by separating nodes based on variability of those attributes, and using a relaxed update rate for most nodes but a more aggressive one for nodes with high variance.

5.1.2 Impact on placement and migration decisions

Next we investigate the impact that error resulting from relaxed measurement intervals would have on the decisions made by a service placement infrastructure. Figures 14 and 15 already contain the answer to this question, as there is an analogy between acceptable migration interval and acceptable data staleness. For example, consider collecting measurements every 30 minutes. If a migration decision is made at the same time as the data is collected, then on average the quality of the sliver-to-node mapping will be the value of the curve at $t = 0$. If a migration decision is made 5 minutes later, but using the same measurements, then the quality of the sliver-to-node matching will be the value of the curve at $t = 5$. And so on, up to $t = 29$. This analogy leads us to a similar conclusion regarding stale data as we made regarding migration interval, namely that data staleness up to 30 minutes is acceptable for making migration decisions for this workload.

5.2 Predicting node resources

In this section, we investigate whether we can predict the availability of a resource on a host using values for other resources on the same host, the same resource on other hosts, or earlier measurements of the same resource on the same host. If such correlations exist, a placement and migration service can reduce measurement overhead by collecting only a subset of the measurements that are needed, and inferring the rest.

5.2.1 Correlation among attributes

We first investigate the correlation among attributes on the same node. A high correlation would allow us to use the value of one attribute on the node to predict the values of other attributes on the node. Table 2 shows the correlation coefficient (r) among attributes on the same node, based on data from all nodes and all timesteps in our trace. Somewhat surprisingly, we see no strong correlations—we might expect to see a correlation between load and network bandwidth, free memory, or swap space. Instead, because each PlanetLab node is heavily multiprogrammed, as suggested by Figure 1, the overall resource utilization is an average (aggregate) across many applications. A spike in resource consumption by one application might occur at the same time as a dip on

resource consumption by another application, leaving the net change “in the noise.” We found a similar negative result when examining the correlation of a single attribute across nodes at the same site (e.g., between load on pairs of nodes at the same site). While we initially hypothesized that there may be some correlation in the level of available resources within a site, for instance because of user preference for some geographic or network locality, the weakness of these same-site correlations implies that we cannot use measurements of a resource on one node at a site to predict values of that resource on other nodes at the site.

r	load	mem	swapfree	bytes_in	bytes_out
load					
mem	-0.04				
swapfree	-0.26	0.18			
bytes_in	0.17	-0.062	-0.20		
bytes_out	0.08	-0.077	0.01	0.44	

Table 2: Correlation between pairs of attributes on the same node: 15-minute load average, free memory, free swap space, 15-minute network receive bandwidth, and 15-minute network transmit bandwidth.

One pair of potentially correlated attributes that merits special attention is inter-node latency and bandwidth. In general, for a given loss rate, one expects bandwidth to vary roughly with $1/\text{latency}$ [16]. If we empirically find a strong correlation between latency and bandwidth, we might use latency as a surrogate for bandwidth, saving substantial measurement overhead.

To investigate the potential correlation, we annotated each pairwise available bandwidth measurement collected by Iperf with the most recently measured latency between that pair of nodes. We graph these (*latency, bandwidth*) tuples in Figure 19. Fitting a power law regression line, we find a correlation coefficient of -0.59, suggesting a moderate inverse power correlation. One reason why the correlation is not stronger is the presence of nodes with limited bandwidth (relative to the bulk of other nodes), such as DSL nodes and nodes configured to limit outgoing bandwidth to 1.5 Mb/s or lower. These capacity limits artificially lower available bandwidth below what would be predicted based on the latency-bandwidth relationship from the dataset as a whole. Measurements taken by nodes in this category correspond to the dense rectangular region at the bottom of Figure 19 below a horizontal line at 1.5 Mb/s, where decreased latency does not correlate to increased bandwidth.

When such nodes are removed from the regression equation computation, the correlation coefficient improves to a strong -0.74. Viewed another way, using a regression equation derived from all nodes to predict available bandwidth using measured latency leads to an average 233% error across all nodes. But if known bandwidth-limited

nodes are excluded when computing the regression equation, predicting available bandwidth using measured latency leads to only an average 36% error across the non-bandwidth-limited nodes. Additionally, certain node pairs show even stronger latency-bandwidth correlation. For example, 48% of node pairs have bandwidths within 25% of the value predicted from their latency. We conclude that a power-law regression equation computed from those nodes with “unlimited” bandwidth (not DSL or administratively limited) allows us to accurately predict available bandwidth using measured latency for the majority of those non-bandwidth-limited nodes. This in turn allows a resource discovery system to reduce measurement overhead by measuring bandwidth among those nodes infrequently (only to periodically recompute the regression equation), and to use measured latency to estimate bandwidth the rest of the time. Of course, if the number of bandwidth-capped nodes in PlanetLab increases, then more nodes would have to be excluded and this correlation would become of less value.

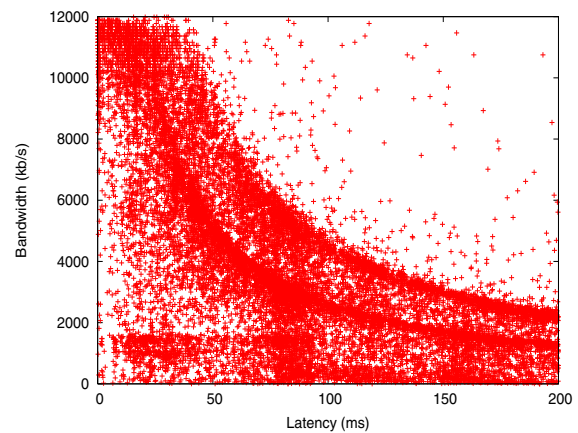


Figure 19: Correlation between latency and available bandwidth. Each point represents one end-to-end bandwidth measurement and the latency measurement between the same pair of nodes taken at the closest time to that of the bandwidth measurement.

5.2.2 Predictability over time

Finally we investigate the predictability of host resources over time. We focus on predicting host 5-minute load average and 15-minute bandwidth average, over periods of 5 minutes and 1 hour.

The most well-known step-ahead predictors in the context of wide-area platforms are those implemented in the Network Weather Service (NWS) [29]. Although originally designed to predict network characteristics, they have also been used to predict host CPU load [31]. We consider the following NWS predictors: last value, exponentially-weighted moving average (EWMA), me-

dian, adaptive median, sliding window average, adaptive average, and running average. For each prediction strategy and host, we compute the average absolute value prediction error for that host across all time intervals, and the standard deviation of the prediction errors for that host. Table 3 shows the average bandwidth prediction error and average load prediction error for the median host using the three NWS techniques that performed best for our dataset (last, EWMA, and median). We also show results for a “dynamic tendency” predictor, which predicts that a series of measurements that has been increasing in the recent past will continue to increase, and a series that has been decreasing in the recent past will continue to decrease [31]. The 5-minute and one-hour predictors operate identically except that the input to the one-hour predictors is the average value over each one-hour period, while the input to the 5-minute predictor is each individual measurement in our trace.

We find that the “last value” predictor performs well over time periods of an hour, confirming and extending our findings from Figures 12 and 13 that load and network bandwidth usage remain relatively stable over periods of an hour. However, as we can see from Figures 2 and 3, the system undergoes dramatic and unpredictable resource demand variations over longer time scales.

The “dynamic tendency” and “last value” predictors perform the best of all predictors we considered, for the following reason. All other predictors (EWMA, median, adaptive median, sliding window average, adaptive average, and running average) predict that the next value will return to the mean or median of some multi-element window of past values. In contrast, the dynamic tendency predictor predicts that the next value will continue along the trend established by the multi-element window of past values. The “last value” predictor falls between these two policies: it keeps just one element of state, predicting simply that the next value will be the same as the last value. PlanetLab load and network utilization values tend to show a mild tendency-based pattern—if the load (or network utilization) on a node has been increasing in the recent past, it will tend to continue increasing, and vice-versa. As a result, the dynamic tendency and last value predictors perform the best. Our results resemble those in [31], which showed errors in the 10-20% range for both last-value and dynamic tendency predictors, in a study of loads from more traditional servers.

Analogous to using a machine’s last load value as a prediction of its next load value, we might use a node’s historical MTTF (MTTR) to predict its future MTTF (MTTR). To evaluate the effectiveness of this technique, we split the trace of node failures and recoveries that we used in Section 3.1 into two halves. For each node, we calculate its MTTF and MTTR during the first half of the trace. We predict that its MTTF (MTTR) during the second half

attribute	prediction technique	5-minute prediction error	1-hour prediction error
load	dynamic tend.	17.8%	23.5%
load	last	17.8%	23.5%
load	EWMA	22.4%	30.7%
load	median	24.3%	35.5%
net bw	dynamic tend.	29.5%	34.6%
net bw	last	29.5%	34.6%
net bw	EWMA	42.7%	48.7%
net bw	median	36.8%	46.2%

Table 3: 5-minute and 1-hour median per-host average prediction error of best-performing NWS predictors and the dynamic tendency predictor.

will be the same as its MTTF (MTTR) during the first half, and calculate the percentage error that this prediction yields. Figure 20 shows a CDF of the fraction of nodes for which this predictor yields various prediction errors. We find substantial prediction error (greater than 100%) for only about 20% of nodes, suggesting that historical node MTTF and MTTR are reasonable criteria for ranking the quality of nodes when considering where to deploy an application. On the other hand, this prediction technique does not yield extremely accurate predictions—for MTTF, error for the median node is 45%, and for MTTR, error for the median node is 87%.

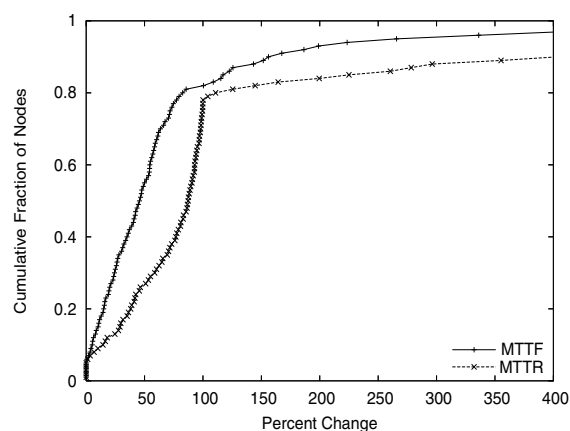


Figure 20: Prediction error for MTTF and MTTR.

Finally, we examine the periodicity of resource availability. Periodicity on the time scale of human schedules is a common form of medium-term temporal predictability. It is often found in utilization data from servers hosting applications with human-driven workloads. For example, a web site might see its load dip when it is nighttime in the time zones where the majority of its users reside, or on weekends. We therefore examined our PlanetLab traces for periodicity of per-node load and network bandwidth usage over the course of a day and week. We found no such periodicity for either attribute. In fact, on average, the load or network bandwidth on a node at time

t was less closely correlated to its value at time $t + 24$ hours than it was to its value at a random time between t and $t + 24$ hours. Likewise, on average, the load or network bandwidth on a node at time t was less closely correlated to its value at time $t + 1$ week than it was to its value at a random time between t and $t + 1$ week.

The lack of daily and weekly periodicity on PlanetLab can be explained by the wide geographic distribution of application deployers and the users of the deployed services such as those studied earlier in this paper. Further, load on PlanetLab tends to increase substantially around conference deadlines, which happen on yearly timescales (beyond the granularity of our trace) rather than daily or weekly ones. In sum, we find that resources values are more strongly correlated over short time periods than over medium or long-term ones.

6. RELATED WORK

The measurement aspects of this paper add to a growing literature on measurements of resource utilization in Internet-scale systems. Most of this work has focused on network-level measurements, a small subset of which we mention here. Balakrishnan examines throughput stability to many hosts from the vantage point of the 1996 Olympic Games web server [2], while Zhang collects data from 31 pairs of hosts [34]. Chen describes how to monitor a subset of paths to estimate loss rate and latency on all other paths in a network [6]. Wolski [29] and Vazhkudai [25] focus on predicting wide-area network resources.

Growing interest in shared computational Grids has led to several recent studies of node-level resource utilization in such multi-user systems. Foster describes resource utilization on Grid3 [8], while Yang describes techniques to predict available host resources to improve resource scheduling [31, 32]. Harchol-Balter investigates process migration for dynamic load-balancing in networks of Unix workstations [12], and cluster load balancing is an area of study with a rich literature. Compared to these earlier studies, our paper represents the first study of resource utilization and service placement issues for a federated platform as heavily shared and utilized as PlanetLab.

Several recent papers have used measurement data from PlanetLab. Yalagandula investigates correlated node failure; correlations between MTTF, MTTR, and availability; and predictability of TTF, TTR, MTTF, and MTTR [30]. Rhea measures substantial variability over time and across nodes in the amount of time to complete CPU, disk, and network microbenchmarks; these findings corroborate our observations in Section 3.1 [19]. Rhea advocates application-internal mechanisms, as opposed to intelligent application placement, to counter node heterogeneity. Lastly, Spring uses measurements of CPU and node availability to dispel various “myths” about PlanetLab [23].

Resource discovery tools are a prerequisite for auto-

mated service placement and migration. CoMon [18], CoTop [17], and Ganglia [14] collect node-level resource utilization data on a centralized server, while MDS [33], SWORD [15], and XenoSearch [22] provide facilities to query such data to make placement and migration decisions.

Shared wide-area platforms themselves are growing in number and variety. PlanetLab focuses on network services [3], Grid3 focuses on large-scale scientific computation [8], FutureGrid aims to support both “eScience” and network service applications [7], and BOINC allows home computer users to multiplex their machines’ spare resources among multiple public-resource computing projects [4]. Ripeanu compares resource management strategies on PlanetLab to those used in the Globus Grid toolkit [21].

7. CONCLUSIONS AND FUTURE WORK

Resource competition is a fact of life when time-shared distributed platforms attract a substantial number of users. In this paper, we argued that careful application placement and migration are promising techniques to help mitigate the impact of resource variability resulting from this competition. We also studied techniques to reduce measurement overhead for a placement and migration system, including using stale or predicted data.

Resource selection and application migration techniques complement the application-specific techniques that some distributed services employ internally to balance load or to select latency-minimizing network paths. Those techniques optimize application performance given the set of nodes already supporting the application, and generally only consider the application’s own workload and structure as opposed to resource constraints due to competing applications. In contrast, this paper focused on *where to deploy—and possibly re-deploy—application instances* based on information about application resource demand and available node and network resources. Once an application’s instances have been mapped to physical nodes, application-internal mechanisms can then be used on finer timescales to optimize performance. In general, application-internal load balancing, external service placement, or a combination of the two can be used to match application instance to available nodes based on resource demand and resources offered.

We expect our observations on placement and migration to generalize to other applications built on top of location-independent data storage; the commonalities we observed among CoDeeN, Coral, and Bamboo, all of which use request hashing in one form or another to determine where data objects are stored, provide initial evidence to support such an expectation. Given the popularity of content-based routing and storage as organizing principles for emerging wide-area distributed systems, this ap-

plication pattern will likely remain pervasive in the near future. A major class of distributed application generally not built in this way is monitoring applications. A monitoring system could store its data in a hash-based storage system running on a subset of platform nodes, making its behavior similar to the applications we examined in this paper (indeed the SWORD system [15] does exactly that). But another common pattern for these applications is to couple workload to location, storing monitoring data at the node where it is produced and setting up an overlay or direct network connections as needed to route data from nodes of interest to the node that issues a monitoring query [13, 27]. In such systems migration is not feasible. Likewise, data-intensive scientific applications that analyze data collected by a high-bandwidth instrument (e.g., a particle accelerator) may wish to couple processing to the location where the data is produced, in which case migration is not feasible. On the other hand, emerging “data grids” that enable cross-site data sharing and federation may reduce this location dependence for some scientific applications, thereby make computation migration more feasible for data-intensive scientific applications in the future.

PlanetLab is the largest public, shared distributed platform in terms of number of users and sites. Thus, we believe that the platform-specific conclusions we have drawn in this paper can extrapolate to future time-shared distributed platform used for developing and deploying wide-area applications that allow users to deploy their applications on as many nodes as they wish and to freely migrate those application instances when desired. A platform with cost-based or performance-based disincentives to resource consumption would likely result in smaller-scale deployments and more careful resource usage, but *variability* in resource utilization across nodes and over time should persist, in which case the usefulness of matching (and re-matching) application resource demand to node resource availability would too.

On the other hand, our “black-box” view of background platform utilization means our results cannot be easily extrapolated to environments that perform global resource scheduling (e.g., all application deployers submit their jobs to a centralized scheduler that makes deployment and migration decisions in a coordinated way), or in which multiple applications make simultaneous placement and migration decisions. Detailed simulation of platform-wide scheduling policies, and the aggregate behavior that emerges from systems with multiple interacting per-application scheduling policies, are challenging topics for future work. Nonetheless, our analysis methodology represents a starting point for evaluating more complex system models and additional placement and migration strategies. As future PlanetLab-like systems such as GENI [9] come online, and as wide-area Grid systems become more widely used, examining

how well these results extrapolate to other environments and application classes will become key research questions.

8. REFERENCES

- [1] A. AuYoung, B. Chun, A. Snoeren, and A. Vahdat. Resource allocation in federated distributed computing infrastructures. In *OASIS '04*, 2004.
- [2] H. Balakrishnan, M. Stemm, S. Seshan, and R. H. Katz. Analyzing stability in wide-area network performance. In *SIGMETRICS*, 1997.
- [3] A. Bavier, L. Peterson, M. Wawrzoniak, S. Karlin, T. Spalink, T. Roscoe, D. Culler, B. Chun, and M. Bowman. Operating systems support for planetary-scale network services. In *NSDI*, 2004.
- [4] Distributed computing: We come in peace. *Red Herring Magazine* <http://www.redherring.com/article.aspx?a=10821>.
- [5] P. Brett. Iperf. <http://www.planet-lab.org/logs/iperf/>.
- [6] Y. Chen, D. Bindel, H. Song, and R. H. Katz. An algebraic approach to practical and scalable overlay network monitoring. In *SIGCOMM*, 2004.
- [7] J. A. Crowcroft, S. M. Hand, T. L. Harris, A. J. Herbert, M. A. Parker, and I. A. Pratt. Futuregrid: A program for long-term research into grid systems architecture. In *Proceedings of the UK e-Science All Hands Meeting*, September 2003.
- [8] I. Foster et al. The grid2003 production grid: Principles and practice. In *HPDC-13*, 2004.
- [9] National Science Foundation. The GENI Initiative. <http://www.nsf.gov/cise/geni/>.
- [10] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with coral. In *NSDI*, 2004.
- [11] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. Sharp: An architecture for secure resource peering. In *SOSP '03*, 2003.
- [12] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *TOCS*, 15(3), 1997.
- [13] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt. Mon: On-demand overlays for distributed system management. In *WORLDS*, 2005.
- [14] M. Massie, B. Chun, and D. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7), 2004.
- [15] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and implementation tradeoffs for wide-area resource discovery. In *HPDC*, 2005.
- [16] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling tcp throughput: A simple model and its empirical validation. In *SIGCOMM*, 1998.
- [17] V. S. Pai. <http://codeen.cs.princeton.edu/cotop/>.
- [18] V. S. Pai. <http://comon.cs.princeton.edu/>.
- [19] S. Rhea, B.-G. Chun, J. Kubiawicz, and S. Shenker. Fixing the embarrassing slowness of opendht on planetlab. In *WORLDS '05*, 2005.
- [20] S. Rhea, B. Godfrey, B. Karp, J. Kubiawicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: A public dht service and its uses. In *SIGCOMM*, 2005.
- [21] M. Ripeanu, M. Bowman, J. Chase, I. Foster, and M. Milenkovic. Globus and planetlab resource management solutions compared. In *HPDC-13*, 2004.
- [22] David Spence and Tim Harris. Xenosearch: Distributed resource discovery in the xenoserver open platform. In *Proceedings of HPDC*, 2003.
- [23] N. Spring, L. Peterson, A. Bavier, and V. S. Pai. Using planetlab for network research: Myths, realities, and best practices. *ACM SIGOPS Operating Systems Review*, 40(1), 2006.
- [24] J. Stribling. All-pairs pings for planetlab. http://www.pdos.lcs.mit.edu/~strib/pl_app/.
- [25] S. Vazhkudai, J. Schopf, and I. Foster. Predicting the performance

- of wide area data transfers. In *IPDPS*, 2002.
- [26] L. Wang, K. Park, R. Pang, V. S. Pai, and L. Peterson. Reliability and security in the codeen content distribution network. In *Usenix Annual Technical Conference*, 2004.
 - [27] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *HotNets II*, 2003.
 - [28] K. Webb, M. Hibler, R. Ricci, A. Clements, and J. Lepreau. Implementing the emulab-planetlab portal: Experience and lessons learned. In *WORLDS '04*, 2004.
 - [29] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1(1), 1998.
 - [30] P. Yalagandula, S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Beyond availability: Towards a deeper understanding of machine failure characteristics in large distributed systems. In *WORLDS*, 2004.
 - [31] L. Yang, I. Foster, and J. M. Schopf. Homeostatic and tendency-based cpu load predictions. In *IPDPS*, 2003.
 - [32] L. Yang, J.M. Schopf, and I. Foster. Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments. In *Supercomputing 2003*, 2003.
 - [33] X. Zhang and J. Schopf. Performance analysis of the globus toolkit monitoring and discovery service, mds2. In *Proceedings of the International Workshop on Middleware Performance (MP 2004)*, April 2004.
 - [34] Y. Zhang, V. Paxson, and S. Shenker. The stationarity of internet path properties: routing, loss, and throughput. Technical report, ACIRI, May 2000.

Replay Debugging for Distributed Applications

Dennis Geels

Gautam Altekar

Scott Shenker

Ion Stoica

University of California, Berkeley

{geels, galtekar, shenker, istoica}@cs.berkeley.edu

Abstract

We have developed a new replay debugging tool, `liblog`, for distributed C/C++ applications. It logs the execution of deployed application processes and replays them deterministically, faithfully reproducing race conditions and non-deterministic failures, enabling careful offline analysis.

To our knowledge, `liblog` is the first replay tool to address the requirements of large distributed systems: lightweight support for long-running programs, consistent replay of arbitrary subsets of application nodes, and operation in a mixed environment of logging and non-logging processes. In addition, it requires no special hardware or kernel patches, supports unmodified application executables, and integrates GDB into the replay mechanism for simultaneous source-level debugging of multiple processes.

This paper presents `liblog`'s design, an evaluation of its runtime overhead, and a discussion of our experience with the tool to date.

1 Introduction

Over the past few years, research has produced new algorithms for routing overlays, query processing engines, byzantine fault-tolerant replication, and distributed hash tables. Popular software like peer-to-peer file sharing applications suggests that interest in distributed applications is not restricted to academic circles.

But debugging is hard. Debugging distributed applications is harder still, and debugging distributed applications deployed across the Internet is downright daunting. We believe that the development of new services has been held back by this difficulty and that *more powerful debugging tools are needed*.

A distributed application is a collection of processes running on machines spread across a network (for our purposes, the Internet). The individual processes may be analyzed independently, and debugging existing tools can catch common “local” errors such as unsafe memory accesses and thread synchronization errors. Unfortunately, these tools do not address the new problems that arise when the processes are composed across an unre-

dictable and lossy network. Races between network messages produce non-deterministic behaviour. Message delay and failure ensure that the aggregate application state is only rarely globally consistent.

Simulation and small-scale test deployments help developers evaluate aggregate system behaviour in a relatively easy environment. With a simulator, the developer has full power to repeat the same execution across multiple experiments, and the state of each application process is available locally for examination. Test deployments complement simulation by adding more realistic network and host machine behaviour. Using local clusters and small, or even emulated, networks, developers may carefully control the degree of realism exposed to their applications.

However, once deployed, distributed applications will reach states that were not tested, and the underlying network will fail in ways that the developer did not anticipate. Long-running services are particularly prone to the slow-developing and non-deterministic, low-probability faults that resist detection during the testing phase.

And once the application is deployed, race conditions and internal state are difficult to observe. Developers rely on application-level logging and `printf` statements, but these techniques only help if the developer chooses to expose the affected internal state before the fault manifests. These types of bugs are generally impossible to reproduce locally, where analysis would be simpler. This *limited visibility* is the core problem for debugging distributed applications. We have developed a new debugging tool, `liblog`, to address it.

1.1 Requirements

We designed this tool to help fix non-deterministic failures in deployed, distributed applications. This goal imposed several requirements on our design.

Deterministic Replay: First and foremost, deployed applications need *logging and replay*. Normal debuggers monitor an application's execution synchronously, so that the process can be paused immediately when a failure, signal, or breakpoint occurs. This approach is infeasible for real, deployed systems for three reasons. First, the latency of a synchronous connection to a re-

mote debugger would significantly slow down the application. Second, pausing the process (or processes, if the developer wished to look at global state) at breakpoints would be unacceptable for real, deployed services, which interact continuously with peer services and clients. Third, real networks are not stable enough to maintain a persistent connection to each process.

Thus debugging must be asynchronous. Each process records its execution to a local log, with sufficient detail such that the same execution can be replayed later. We should follow the same code paths during replay, see the file and network I/O, and even reproduce signals and other IPC. The replay could run in parallel with the original execution, after the original process dies, or even on a completely different machine.

Continuous Logging: In order to record the manifestation of slow-developing and non-deterministic, low-probability faults, the logging infrastructure must remain active at all times. We must operate under the assumption that more bugs are always waiting. Also, any slight perturbations in application behaviour imposed by the debugger becomes the “normal” behaviour. Removing it then would be a perturbation that might activate so-called “heisenbugs”.

If the debugging system required significant resources, the cost in performance (or faster hardware) might be prohibitive. Fortunately, many types of distributed applications consume relatively few local resources themselves. Whereas network bandwidth and latency might be precious, we often have extra CPU cycles and disk space to accommodate our logging tools. In particular, if we confine ourselves to a small processing budget, the network will remain the performance bottleneck, and the application will exhibit little slowdown.

Consistent Group Replay: We are particularly interested in finding *distributed bugs*, such as race conditions and incorrect state propagation. This kind of error may be difficult or impossible to detect from the state of any one process. For example, transient routing loops are only visible when the aggregate state of multiple routers is considered.

So we must be able to see snapshots of the state across multiple processes and to trace message propagation from machine to machine. Naturally, true snapshots are impossible without synchronized clocks (cf. [Lam78]), but we can require that each machine is replayed to a *consistent* point, where no message is received before it has been sent.

Mixed environment: Most applications will not run our software, particularly client software and supporting services like DNS. This fact becomes a problem if we require coordination from communication peers during logging or replay, as we generally must in order to sat-

isfy the previous requirement (consistent replay). Since we do not operate in a closed system, our tools must understand the difference between cooperating and non-cooperating peers and treat each appropriately.

1.2 Contributions

The primary contribution of our work is the design and evaluation of a debugging tool, `liblog`, that satisfies each of these requirements. Previous projects have developed logging and replay tools that focus on either low overhead or providing consistent replay, but we have addressed both. Furthermore, to the best of our knowledge, `liblog` is the first tool that (1) provides consistent replay in a mixed environment, or (2) allows consistent replay for arbitrary subsets of application processes.

In addition, `liblog` requires neither special hardware support nor patches to privileged system software. Also, it operates on unmodified C/C++ application binaries at runtime, without source code annotations or special compilation tools. Multithreading, shared memory, signals, and file and network I/O all work transparently.

Finally, we designed `liblog` to be simple to use. Logging only requires running our start-up script on each machine. Our replay tools make debugging as easy as using GDB for local applications: they automate log collection, export the traditional GDB interface to the programmer, and even extend that interface to support consistent replay of multiple processes and tracking messages across machines.

We built `liblog` by combining existing technology in new ways and extending the state of the art as necessary. In the following sections, we will present an overview of the resulting design (Section 2) and then explain in more detail the new technical challenges that arose, along with our solutions (Section 3).

1.3 Is liblog Right For You?

We designed `liblog` with lightweight distributed applications like routing overlays in mind. We assume that the host machines have spare resources—specifically CPU, memory, network, and disk—that we can apply to our debugging efforts.

Although it can correctly log and replay general C/C++ applications, the runtime overhead imposed could outweigh the benefits for resource-intensive systems like streaming video servers or heavily multithreaded databases. We quantify this overhead in Section 4.

2 Design

In this section we present an overview of `liblog`’s design, highlighting the decisions that we made in order to satisfy the requirements listed above.

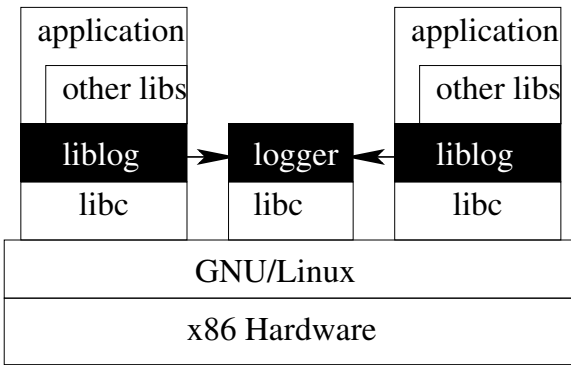


Figure 1: *Logging*: liblog intercepts calls to libc and sends results to logger process. The latter asynchronously compresses and writes the logs to local storage.

2.1 Shared Library Implementation

The core of our debugging tool is a shared library (the eponym liblog), which intercepts calls to libc (e.g., select, gettimeofday) and logs their results. Our start-up scripts use the LD_PRELOAD linker variable to interpose liblog between libc and the application and its other libraries (see Figure 1). liblog runs on Linux/x86 computers and supports POSIX C/C++ applications.

We chose to build a library-based tool because operating in the application’s address space is efficient. Neither extra context switches nor virtualization layers are required. Alternative methods like special logging hardware [NM92, XBH03, NPC05] or kernel modifications [TH00, SKAZ04] can be even faster, but we found these solutions too restrictive for a tool that we hope to be widely adopted and deployed.

Another promising alternative is to run applications on a virtual machine and then to log the entire VM [KDC05, SH, HH05]. We rejected it because we believe that VM technology is still too difficult to deploy and too slow for most deployed services.

On the other hand, there are serious drawbacks of a library implementation. First, several aspects of observing and controlling applications are more difficult from within the address space, most notably supporting multiple threads and shared memory. We will discuss these challenges in Section 3.

Fundamentally, however, operating in the application’s address space is neither complete (we cannot replay all non-determinism) nor sound (internal state may become corrupted, causing mistakes). We will discuss such limitations in Section 4.

Nevertheless we believe that the combined efficiency and ease of use of a library-based logging tool makes this solution the most *useful*.

2.2 Message Tagging and Capture

The second defining aspect of our logging tool is our approach to replaying network communication. We *log the contents* of all incoming messages so that the receiving process can be replayed independently of the sender.

This flexibility comes at the cost of significant log space (cf. Section 5) but is well justified. Previous projects have tried the alternative, replaying *all* processes and regenerating message contents on the sender. We cannot do so because we operate in a mixed environment with non-logging processes. Even cooperating application logs may be unavailable for replay due to intervening disk or network failure.

So far we satisfy one requirement, but we must be able to coordinate these individual replays in order to provide another, Consistent Group Replay. For this purpose, we embed 8-byte Lamport clocks [Lam78] in all outgoing messages during execution and then use these virtual clocks to schedule replay. The clock update algorithm ensures that the timestamps in each log entry respect the “happens-before” relationship. They also provide a convenient way to correlate message transmission and reception events, so we can trace communication from machine to machine.

To make the virtual clocks more intuitive, we advance them at the rate of the local machine clock. If the machine clocks happen to be synchronized to within one network RTT, the virtual clocks will match exactly.

2.3 Central Replay

Our third major design decision was to enable off-site replay. Rather than restart each process *in situ*, a central console automatically downloads the necessary logs and checkpoints and instantiates each replay process locally. Local replay removes the network delay from the control loop, making it feasible to operate on distributed state and to step across processes to follow messages.

The costs are several: first, the network bandwidth consumed by transferring logs may exceed that required to control a remote debugger. Second, the hardware and system software on the replay machine must match the original host; currently we support only GNU/Linux/x86 hosts. Third, we must log data read from the local file system (as with network messages) because the files may not be available on the replay machine. This technique also obviates maintaining a versioned file system or undoing file modifications. Finally, building a migratable checkpoint system is challenging. We consider the first two costs to be acceptable and will discuss our solution to the last challenge in Section 3.6.

3 Challenges

In this section we will discuss the technical challenges we faced when building our logging and replay system. Most are new problems caused by our user-level implementation and/or message annotations; previous projects did not address them because their focus allowed for different design choices.

3.1 Signals and Thread Replay in Userland

As we noted earlier, logging and replaying applications at the `libc` level assumes that they only interact with their environment through that interface and that, outside of `libc` calls, the application execution is deterministic. This assumption fails when multiple threads execute concurrently on the same address space. The value read from a shared variable depends on the order in which competing threads modify it; every write could be a race condition. The same problem arises when multiple processes share memory segments or when signal handlers (effectively another thread) access global variables.

To make replay deterministic in these cases, we must either intercept and replay the *value* of each read from shared memory, or we must replay each read and write in the same *order*, so races resolve identically. The former option is too invasive and requires log bandwidth proportional to the memory access stream. The latter is still expensive, but the cost can be reduced significantly by logging only the order and timing of thread context switches. If we assume a single processor, or artificially serialize thread operation, then identical thread schedules produce identical memory access patterns.

The challenge in our case was to record and replay thread schedules using only our user-level shared library. The task is relatively simple for kernel- or VM-based tools, but user-level libraries generally have no ability even to observe context switches among kernel threads, much less control them. We believe that `liblog` is the first to address the problem.

Our solution effectively imposes a user-level cooperative scheduler on top of the OS scheduler. We use a pthread mutex to block all but one thread at a time, ignoring conflicting context switches by the kernel. The active thread only surrenders the lock at `libc` call points, as part of our logging wrapper, and the next active thread logs the context switch before continuing. Processes that share memory are handled identically. Similarly, signals are queued and delivered at the next `libc` call.

Restricting context switches to our wrapper functions provides a convenient point to repeat the switches during replay, but the change to thread semantics is not fully transparent. In particular, we cannot support applications that intentionally use tight infinite loops, perhaps as part of a home-grown spin lock, because other threads will

not have any opportunity to acquire our scheduling lock. Delaying signals may affect applications more, although we note that the kernel already tries to perform context switches and to deliver signals at syscall boundaries, so the impact of our solution may not be pronounced. We have not yet quantified the degree to which the schedule we impose differs from a normal one.

3.2 Unsafe Memory Access

Another potential source of non-determinism arises when an application reads from uninitialized (but allocated) heap memory or beyond the end of the stack. The contents of these memory regions are not well defined for C applications, and in practice they change between execution and replay. One could argue that accessing these regions could be considered incorrect behaviour, but it is legal, reasonably safe, and present even in robust software like OpenSSL [SSL].

Much of the change in memory between logging and replay is due to the logging tool itself, which calls different functions during replay, leaving different stack frames and allocating different memory on the heap. One can significantly minimize the tool's memory footprint, as stressed in Jockey [Sai05], but it can never be completely eliminated by a library-based debugging tool. Internal memory use by `libc` will always differ because its calls are elided during replay, so `malloc` may return different memory to the application.

Our solution is simpler: we merely zero-fill all memory returned by `malloc` (effectively replacing it with `calloc`) as well as stack frames used by our `libc` wrappers. Thus, uninitialized reads replay deterministically, even if `malloc` returns a different region. This solution still fails if the application depends on the actual address, for example, as a key for a hash table.

Also, it is very difficult to protect a library-based tool from corruption by stray memory writes into the tool's heap. A virtual machine-based alternative would avoid this problem. Also, one could imagine disabling write access to the `liblog`'s memory each time control returns to the application. Instead, we rely on dedicated memory-profiling tools like Purify [Pur] and Valgrind [Val] to catch these various memory errors, so that we can focus on efficient logging.

3.3 Consistent Replay for TCP

As described in Section 2.2, we annotate all network messages between application processes with Lamport clocks so that we can replay communicating peers consistently. For datagram protocols like UDP, we use simple encapsulation: we prepend a few bytes to each packet, and remove them on reception. We pass a scatter/gather array to `sendmsg` to avoid extra copies.

Annotating byte streams like TCP is more compli-

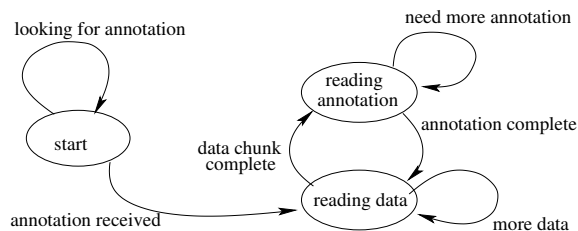


Figure 2: *Receiving Annotated TCP*: Detecting and extracting Lamport clocks from incoming byte streams requires additional bookkeeping.

cated, because timestamps must be added throughout the stream, at the boundary of each sent data chunk. But the receiver need not consume bytes in the same batches; it often will read all available data, be it more or less than the contents of a single send payload.

Our solution is a small (3-state) state machine for each incoming TCP connections (see Figure 2). Once the stream has been verified as containing annotations, the state machine alternates reading annotations and reading application data until the calling function has enough data or the socket is drained. Each state transition requires a separate call to read the underlying stream; we cannot simply read extra bytes and extract the annotations, because we cannot anticipate how far to read. We do not know the frequency of future annotations, and attempting to read more data than necessary may cause the application to block needlessly. It is always possible that more bytes will not arrive.

If multiple annotations are consumed by a single read call, we log the most recent timestamp, as it supersedes the others. Naturally, we remember the stream state between calls so that we may continue even if the last read attempt ended in the middle of an annotation.

3.4 Finding Peers in a Mixed Environment

Embedding annotations in messages also complicates interaction with non-logging processes such as third-party clients, DNS and database servers, or, if `liblog` is only partially deployed, even fellow application processes. These non-loggers do not expect the annotations and would reject or (worse yet) misinterpret the message. We believe that this problem is the reason that no previous logging tool has supported consistent replay in a mixed environment.

We must either send annotations that will be safely ignored by non-logging processes or discover whether a remote peer expects annotations and omit them when appropriate. The former option could be implemented using either IP options¹ or the out-of-band (OOB) channel for TCP connections, but either method would conflict with

¹See RFC 791

networks that already used these paths. Also, we have seen evidence that adding IP options has a negative impact on application traffic, and OOB does not help UDP traffic (nor incompatible TCP implementations).

We opted for a safer, but slower, solution. The logger on each machine tracks the local ports opened by logging processes and listens on a globally well-known port (currently 5485). This approach fails to fully support applications hidden behind NAT-enabled firewalls, but it could easily be replaced by a more sophisticated discovery mechanism. Each `liblog`-enabled process then queries the remote logger (via TCP) before sending the first datagram or opening a TCP connection. The query contains the destination port and protocol of interest and asks whether that port is currently assigned to a logging process.

If the application receives a negative reply, or none at all, that packet flow will not be annotated. Replies are cached for the duration of a stream, or 30 seconds for datagram sockets, to amortize the query latency overhead. Currently, we wait a maximum of 2 seconds for a query, but that maximum is only reached if the remote machine has no logger and does not reset our TCP request. But this case does happen frequently for firewall-protected machines, so we cache information on dropped queries for up to 5 minutes.

3.5 Replaying Multiple Processes

The real power of replay debugging depends on the ability to set breakpoints, to pause execution, and to observe internal application state, just as one can in normal debuggers. Rather than develop new technology with its own interface, we decided to adapt the GNU debugger [GDB]. GDB provides a powerful and familiar interface for controlling application execution and accessing internal state by symbolic names.

Unfortunately, GDB, like many debuggers, can only control a single process. Replaying multiple processes, or even children created with `fork`, requires multiple instances of the debugger. Our challenge was to coordinate them, multiplexing their input and output to the programmer and scheduling the application execution so that replay is consistent.

We use a two-tiered approach to controlling the replay processes. Threads within a process group are multiplexed by the same scheduling locks used during logging (cf. Section 3.1), always choosing the next thread based on the schedule stored in the log. These locks also block a newly `fork`-ed process until we attach a new GDB instance to it.

Across process groups, consistent replay is enforced by our replay console, a small Python [Py] application. For each application process, the console uses GDB to set breakpoints in key `libreplay` functions. These

pause execution at each `libc` call, allowing us to schedule the next process or to download the next set of logs.

The replay console provides a single interface to the programmer, passing commands through to GDB and adding syntax for broadcasting commands to multiple processes. It also allows advanced programmability by interacting directly with the underlying Python interpreter.

3.6 Migratable Checkpoints

Replaying application processes centrally, offline, makes the debugger more responsive and makes it feasible to operate on distributed application state. But restarting processes on a new machine is tricky. The two main challenges are first, to copy the state of the original application into a live process on the new machine, and second, to reconcile this new process with the debugger (GDB).

Our checkpoint mechanism is based on the `ckpt` [Ckp] library from the University of Wisconsin. This library reads the `/proc/` filesystem to build a list of allocated memory regions for the application and then writes all such memory to a checkpoint file. For replay, a small bootstrap application reads that file and overwrites its own memory contents, adjusting memory allocations as necessary.

First we extended `ckpt` to handle the kernel-level thread state for multi-threaded applications, which was simplified by our user-level scheduler. A thread saves its state before relinquishing the CPU, so at any time we have the state of all inactive threads stored in our tables.

Next we added support for shared memory regions: each process in a group checkpoints its private memory, and one “master” process writes and restores the shared memory for everyone.

Integrating checkpoint support to GDB required additional work. Starting the process within GDB is problematic because the symbol tables of the bootstrap program and the restored application do not generally agree, or even necessarily overlap, and GDB does not support symbol tables moving during runtime. Even if we use the original application to bootstrap the process, GDB becomes confused when shared libraries are restored at new locations.

To solve this problem, we added a new method for finding the in-memory symbol table of a running application (by reading the `r_debug.r_brk` field), ignoring the conflicting information from the local executable file. It is then sufficient to attach to the restored application and to invoke this new symbol discovery method.

Our modifications required adding approximately 50 lines of code, including comments, to one source file in GDB. Most of those lines comprise the new function for locating the symbol table.

4 Limitations

There are several limitations to our debugging tool, both fundamental and mundane.

Log storage The biggest reason for a developer to *not* use `liblog` with an application is the large amount of log data that must be written to local disk. Log storage is a fundamental problem for any deterministic replay system, but our approach to handling I/O (cf. Section 2) renders `liblog` infeasible for high-throughput applications. Every Megabyte read from the network or disk must be logged (compressed) to the local disk, consuming space and disk bandwidth. This approach is acceptable for relatively lightweight applications like routing overlays, consuming only a few megabytes per hour, but is probably unrealistic for streaming video or database applications. We will quantify the problem in Section 5.

Host requirements Our basic logging strategy only addresses POSIX applications and operating systems that support run-time library interposition. In practice, our OS options are restricted even further, to recent Linux/x86 kernels (2.6.10+) and GNU system software (only `libc` 2.3.5 has been tested). These limitations are imposed by our borrowed checkpointing code and compatibility issues with our modified version of GDB.

Scheduling semantics As explained in Section 3.1, `liblog`’s user-level scheduler only permits signal delivery and context switches at `libc` function calls. The OS generally tries to do the same, so most applications will not notice a significant difference.

However, we are assuming that applications make these calls fairly regularly. If one thread enters a long computation period, or a home-grown spin lock implemented with an infinite loop, `liblog` will never force that thread to surrender the lock, and signals will never be delivered. We are exploring solutions to this problem.

Network overhead Our network annotations consume approximately 16 bytes per message, which may be significant for some applications. The first 4 bytes constitute a “magic number” that helps us detect incoming annotations, but this technique is not perfect. Thus another limitation is that streams or datagrams that randomly begin with the same sequence of 4 bytes may be incorrectly classified by `liblog` and have several bytes removed. This probability is low (1 in 2^{32} for random messages), and is further mitigated by additional validity checks and information remembered from previous messages in a flow, but false positives are still possible.

Limited consistency Fundamentally, consistent replay in a mixed environment is not guaranteed to be perfectly consistent. A message flow between two application processes loses its timing information if the flow is relayed

by a non-logging third party. Then, if the virtual clocks for the two processes are sufficiently skewed, it is possible to replay message transmission *after* its reception. The probability of this scenario decreases rapidly as the application's internal traffic patterns increase in density, which keeps the virtual clocks loosely synchronized.

Completeness Finally, as mentioned earlier, library-based tools are neither complete nor sound, in the logical sense of the words. They are incomplete because they cannot reproduce every possible source of non-determinism. `liblog` addresses non-determinism from system calls, from thread interaction, and, to a lesser extent, from unsafe memory accesses. Jockey [Sai05] focuses on a different set of sources, reducing changes to the heap and adding binary instrumentation for intercepting non-deterministic x86 instructions like `rdtsc` and, potentially, `int`.

Unfortunately, logging libraries will never succeed in making the replay environment exactly identical to the original environment because they operate inside the application's address space. The libraries run different code during logging and during replay, so their stack and heap differ. Theoretically, an unlucky or determined application could detect the difference and alter its behaviour.

Soundness We say logging libraries are *unsound* because, as part of the application, they may be corrupted. We hope that applications have been checked for memory bugs that could cause stray writes to `liblog`'s internal memory, but C is inherently unsafe and mistakes may happen. We do assume the application is imperfect, after all.

Furthermore, libraries are susceptible to mistakes or crashes by the operating system, unlike hardware solutions or virtual machines (although even virtual machines generally rely on the correctness of a *host* OS).

Fortunately, these theoretical limitations have little practical impact. Most applications are simple enough for `liblog` to capture all sources of non-determinism, and simple precautions to segregate internal state from the application's heap are usually sufficiently safe. Indeed, most debuggers (including GDB) are neither sound nor complete, but they are still considered *useful*.

5 Evaluation

We designed `liblog` to be sufficiently lightweight so that developers would leave it permanently enabled on their applications. In this section, we attempt to quantify the overhead imposed by `liblog`, both to see whether we reached this goal and to help potential users estimate the impact they might see on their own applications.

We start by measuring the runtime latency added by our `libc` wrappers and its effect on network performance. TCP throughput and RTT are not noticeably af-

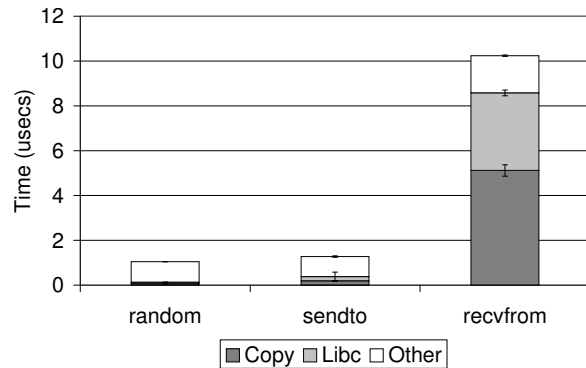


Figure 3: *Wrapper Overhead*: time required to intercept and log `libc` functions. The *copy* region measures the time taken to write the bytes to a shared memory region monitored by the logger, and *other* includes the overhead of intercepting the calls and our internal bookkeeping. The *libc* region measures the time taken for the underlying library call to complete.

ected. A second set of experiments measures the storage overhead consumed by checkpoints and logs.

All experiments were performed on a Dual 3.06GHz Pentium 4 Xeon (533Mhz FSB) with 512K L2 cache, 2GB of RAM, 80GB 7500 rpm ATA/100 disk, and Broadcom 1000TX gigabit Ethernet.

5.1 Wrapper Latency

To measure the processing overhead of `liblog`, we first analyzed the latency added to each `libc` call. Figure 3 shows the latency for a few representative wrappers.

The wrappers add approximately 1 microsecond to the function `random`, which shows the minimum amount of work each wrapper must do to intercept the call and to write a log entry. The `sendto` wrapper is slightly slower as it includes the amortized cost of querying the destination to determine whether to send annotations (cf. Section 3.4). The “copy” phase is also longer, because we store the outgoing message address and port to facilitate message tracing. The `recvfrom` overhead is higher still because it must extract the Lamport clock annotation from the payload and copy the message data to the logs.

5.2 Network Performance

Next we measured the impact of `liblog` on network performance. First we wrote a small test application that sends UDP datagrams as fast as possible. Figures 4 and 5 show the maximum packet rate and throughput for increasing datagram sizes. With `liblog` enabled, each rate was reduced by approximately 18%.

For TCP throughput, we measured the time required for `wget` to download a 484 MB binary executable from various web servers. Figure 6 shows that `liblog` hinders `wget` when downloading the file over a gigabit

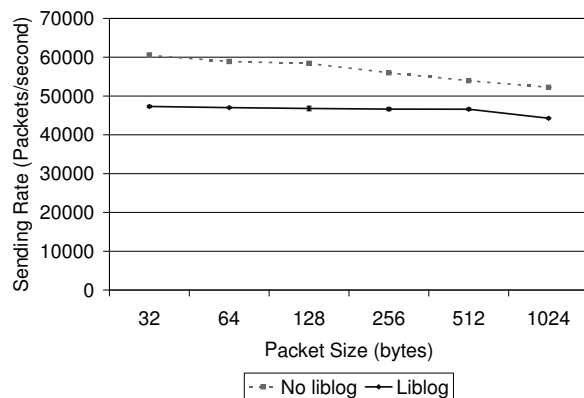


Figure 4: *Packet rate reduction*: Maximum UDP send rate for various datagram sizes. The maximum standard deviation over all points is 1.3 percent of the mean.

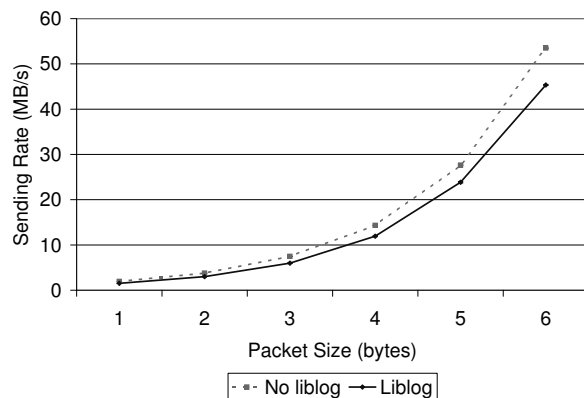


Figure 5: *UDP bandwidth*: Maximum UDP send throughput for various datagram sizes. The maximum standard deviation over all points is 1.3 percent of the mean.

ethernet link, but the reduction in throughput is negligible when the maximum available throughput is lowered. Even the relatively fast 100 MBps link to our departmental web server can be filled using `liblog`.

Finally, Figure 7 shows the round-trip time (RTT) measured by `lmbench` to the local host and to a machine on a nearby network. The gigabit ethernet test shows that `liblog` adds a few wrappers worth of latency to each RTT, as expected. On a LAN, the RTT overhead is so small that the difference is hard to discern from the graph.

5.3 Log Bandwidth

The amount of log space required depends greatly on the frequency of `libc` calls made by an application, as well as on the throughput and content of its network traffic, because incoming message contents are saved.

To give an idea of the storage rates one might expect,

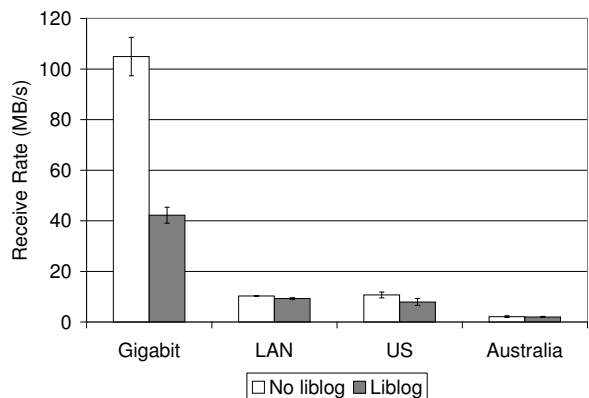


Figure 6: *TCP throughput* for `wget` downloading a 484MB file. Each pair of bars represents a different web server location.

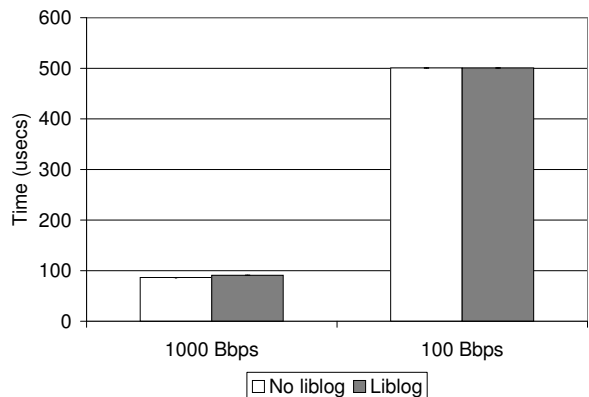


Figure 7: *RTT overhead*: measured by `lmbench`. The error bars cannot be seen in these graphs because the standard deviation is negligible.

we first measured the average log growth rate of the applications we use ourselves: `I3/Chord` and the `OCALA` proxy. For this experiment, we started a small `I3` network on PlanetLab and attached a single local proxy. No additional workload was applied, so the processes were only sending their basic background traffic. We also show the logging rates for `wget` downloading an executable file when we artificially limit its download rate to simulate applications with various network throughput. Figure 8 shows the (compressed) log space required per hour for each application. This rate varies widely across applications and correlates directly with network throughput. We have found the 3-6 MB/hour produced by our own applications to be quite manageable.

Figure 9 illustrates the degree to which message contents affect the total log size. We limited `wget` to a 1 KB/s download rate and downloaded files of various entropy. The first file was zero-filled to maximize compressibility. Then we chose two real files: File A is a

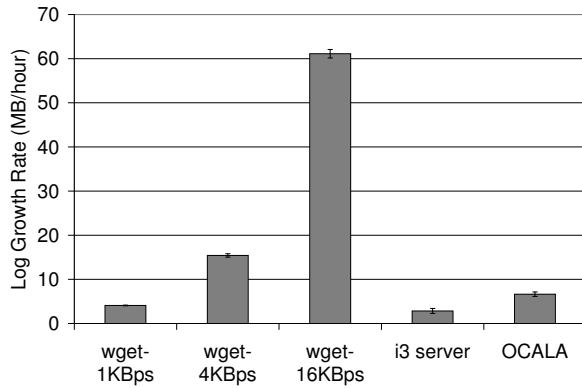


Figure 8: *Log bandwidth*: Log size written per hour for various applications. The bottom three columns correspond to wget with the specified cap on its download rate.

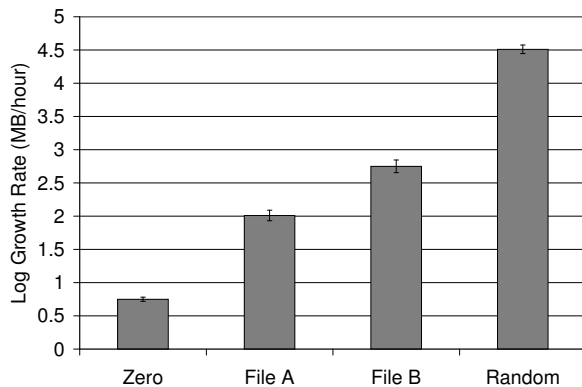


Figure 9: *Log entropy*: Log size written by wget depends on compressibility of incoming data.

binary executable and File B is a `liblog` checkpoint. Finally, we try a file filled with random numbers, which, presumably, is incompressible. The difference between zero and full entropy is over an order of magnitude, although most payloads are presumably somewhere in the middle.

5.4 Checkpoint Overhead

Finally, we measured the checkpoint latency (Figure 10) and size (Figure 11) for a few of our test applications. The checkpoint size depends on the amount of the application's address space that is in use. The checkpoint latency is dominated by the time required to copy the address space to file system buffers, which is directly proportionally to the (uncompressed) checkpoint size. These costs can be amortized over time by tuning the checkpoint frequency. The trade-off for checkpoint efficiency is slower replay, because more execution must be replayed on average before reaching the point of interest.

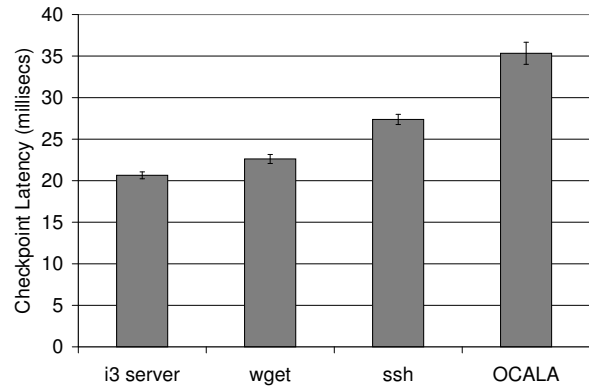


Figure 10: *Checkpoint Latency*: time taken to dump memory to checkpoint file for various applications.

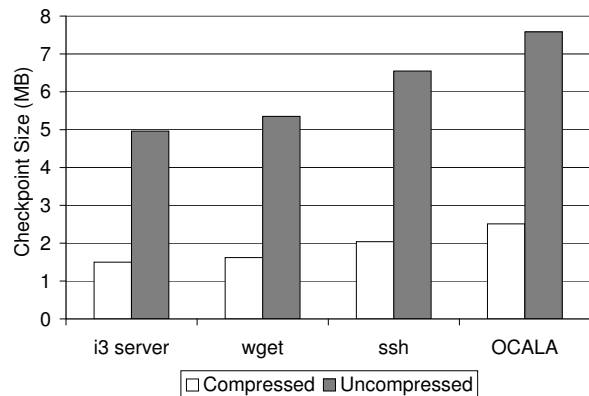


Figure 11: *Checkpoint Size*: total and compressed size of checkpoints for various applications.

5.5 Evaluation Summary

These experiments suggest that the CPU overhead imposed by `liblog` is sufficiently small for many environments and has little affect on network performance. Logging could consume considerable disk space (and disk bandwidth), but the distributed applications we are familiar with (I3/Chord and OCALA) could store logs for a week or two, given 1GB of storage. Checkpoints also consume a noticeable amount of space, but writing one once an hour is probably sufficient for most cases.

6 Experience

We have been working on `liblog` for over a year, but we completed the prototype described in this paper only a few weeks ago. In the intervening time, we have used the tool on distributed applications with which we are familiar, namely I3/Chord [SAZ⁺02] and the OCALA proxy [JKK⁺06]. We have already discovered several errors in these applications. In this section, we will de-

scribe how `liblog` helped in these cases, along with a few stories from earlier prototypes and work debugging `liblog` itself.

6.1 Programming Errors

To start, we found a few simple mistakes that had escaped detection for months. The first, inserted accidentally by one of this paper's authors over a year ago, involved checking Chord timeouts by calling `gettimeofday` within a "MAX" macro that evaluated its arguments twice. The time changed between calls, so the value returned was not always still the maximum.

We also found an off-by-one error in code that assumed 1-based arrays and timer initialization code that did not add `struct timeval` microseconds properly, both in OCALA's I3 library.

The off-by-one error normally had no visible effect but occasionally caused the proxy to choose a distant, high-latency gateway. The two timer-related errors only manifested occasionally but would cause internal events to trigger too late, or too early, respectively.

These bugs had escaped earlier testing because they were non-deterministic and relatively infrequent. But once we noticed the problems, `liblog` was able to deterministically replay the exact execution paths so that we could step through the offending code in GDB and watch the problem unfold.

6.2 Broken Environmental Assumptions

Perhaps more interesting are bugs caused not by programmer mistakes but rather by correct implementation based on faulty assumptions. To illustrate, here are two problems in Chord we had found with an earlier `liblog` prototype.

The first problem is common in peer-to-peer systems, and was discussed along with solutions in a later paper [FLRS05]. Basically, many network overlays like Chord assume that the underlying IP network is fully connected, modulo transient link failures. In practice, some machine pairs remain permanently disconnected due to routing policy restrictions and some links experience unexpected partial failure modes, such as transient asymmetry. Both problems cause routing inconsistencies in Chord, and both were witnessed by `liblog` in a network deployed across PlanetLab [PL].

Rather than finding a coding error in the application, replay showed us code that worked as designed. Our project is focused on application debugging, and we do not attempt to debug the underlying network; nevertheless, our logs clearly showed the unexpected message-loss patterns. Of course the problem had not been detected using simulation, because the simulator made the same assumptions about the network as the application.

A second assumption we had made was that our appli-

cation processes would respond to keep-alive messages promptly. Chord includes RTT estimation and timeout code based on TCP, which expects a reasonable amount variance. On PlanetLab, however, high CPU load occasionally causes processes to freeze for several seconds, long enough for several successive pings to time out. Chord then incorrectly declared peers offline and potentially misrouted messages.

Upon inspection, `liblog` showed us that the timeout code was operating correctly, and the message tracing facilities detected the keep-alive responses arriving at the correct machines, although long after they had been considered lost. The virtual clock timestamps let us correlate otherwise-identical messages, as well as detect the long delay in between system calls on the pinged machine.

6.3 Broken Usage Assumptions

We found two problems with the OCALA proxy's overlay client initialization code, both caused by sensitivity to the bootstrap gateway list. Like those of the previous section, these "bugs" were not programming errors per se, but rather user errors (providing an imperfect configuration list) or design flaws (not tolerating user error).

One phase of startup involves pinging these gateways and triangulating the local machine's latitude and longitude based on the response times. We noticed that the proxy occasionally made a very poor estimate of local coordinates, which then caused a poor (high latency) choice of primary gateways.

We investigated the phenomenon by setting breakpoints in the relevant methods and stepping through the replay. We noticed first that very few points were used for triangulation. We then moved *backwards* in the execution to find that only a small number of pings were sent and that the proxy did not wait long enough for most the replies. If care is taken to nominate only lightly loaded gateways, triangulation works fine. If not, as in our case, performance suffers until periodic maintenance routines manage to choose a better gateway, which could take hours.

We also discovered that the proxy client is very trusting of liveness information contained in the initial gateway list. Normally this list is continually updated by an independent process so that only active gateways are included. If the list becomes stale, as we unintentionally allowed, the proxy could waste minutes trying to contact dead I3 servers before finally connecting.

We diagnosed the problem by replaying and comparing the paths taken by two executions: one which exhibited the interminable timeouts and one which lucked upon a good subset of gateways immediately. This problem could easily be dismissed as invalid usage. Nevertheless, solving it relied on our ability to deterministically replay the random choices made during the gateway se-

lection process.

6.4 Self-Debugging

The program we have spent the most time debugging recently is `liblog` itself. Because the tools run as shared libraries in the application address space, we are able to use GDB to set breakpoints and to step through our own code during replay, just like the supposed target application. We used this ability to fix programming errors in our message annotation layer and our remote discovery service. Deterministic replay also made it easy to find faults in our replay console because each log provided a repeatable test case.

Some bugs in `liblog`, such as incomplete `libc` wrappers, manifest as non-determinism during replay. Ironically, this non-determinism made them easy to detect because we could step through the execution at the point where the original execution and replay diverged in order to isolate the failure. This approach also led us to realize the problem of applications accessing undefined heap and stack memory.

6.5 Injected Bugs

Our tool is interactive, aiding a human programmer but requiring their domain knowledge and expertise. We find it difficult to quantify the benefit `liblog` provides because the user injects a large amount of variability into the process. Ideally, we will be able to compile a large library of “real” bugs that exist in tested and used applications for some time before being fixed with `liblog`. But this process is slow and unpredictable.

Projects that develop automated analytic techniques often pull known errors from bug databases and CVS histories in order to quantify how many of the problems can be re-fixed with their tools. This path is also available to use, but the results would be somewhat suspect as the human tester may have some prior knowledge of old bugs. Similar doubts may arise if one set of programmers manually introduces errors into a current application code base for testing by an independent second group. This trick has the benefit of testing our tools on bugs that are arbitrarily complex or slow to develop.

While we wait for our library of real bugs to grow, we have decided to try both of these somewhat-artificial testing methods. So far we have only started on the latter, with one author injecting an error into the I3/Chord code base while the other uses `liblog` to isolate and fix it. Preliminary results suggest that the task is equivalent to debugging Chord in a local simulator. We plan to have more results in this vein soon.

7 Related Work

Deterministic replay has been a reasonably active research subject for over two decades. Most of this work

centered on efficient logging for multiprocessors and distributed shared memory computers; for an overview of the field we recommend an early survey by Dionne et al [DFD96] and later ones by Huselius [Hus02] and Cornelis et al [CGC⁺03].

None of these previous projects focused on deployed, distributed applications or addressed the technical challenges raised by that set of requirements. In particular, our support of consistent group replay in a mixed environment is unique, and we are the first to address the challenges described in Section 3, such as supporting multithreaded applications without kernel support.

On the other hand, the core techniques of logging and replay have been explored thoroughly, and we borrowed or reinvented much from earlier projects. Specifically, Lamport clocks [Lam78] have been used for consistent replay of MPI [RBdK99] and distributed shared memory [RZ97]. Replaying context switches to enforce deterministic replay in multithreaded apps was based on DeJaVu [KSC00], which built the technique into a Java Virtual Machine. Finally, some projects have integrated GDB and extended its interface to include replay commands [SKAZ04, KDC05], but only `liblog` seamlessly provides consistent replay across multiple processes.

Our library-based implementation most closely resembles Jockey [Sai05]; they also have simple binary-rewriting functionality to detect use of non-deterministic applications. Flashback [SKAZ04] also has many similarities, but they chose to modify the host OS. Their modifications enable very efficient checkpoints and (potentially) simplified thread support. We chose instead to implement all of `liblog` at user level in order to maximize its portability and to lower barriers to use on shared infrastructure. Also, our support for multiple threads, migratable checkpoints, and consistent replay across machines makes `liblog` more appropriate for distributed applications.

The DeJaVu project [KSC00] shared our goal of replaying distributed applications. Like `liblog`, they support multithreaded applications and consistently replay socket-based network communication. Unlike `liblog`, they targeted Java applications and built a modified Java Virtual Machine. Thus they addressed a very different set of implementation challenges. Also, they do not support consistent replay in a mixed environment, although they do sketch out a potential solution.

8 Conclusion

We have designed and built `liblog`, a new logging and replay tool for deployed, distributed applications. We have already found it to be useful and would like to share the tool with others in the distributed systems community. A software distribu-

tion package and more information is available at <http://research.geels.org:8080/>.

We have plans for a few additional improvements to liblog, both to reduce its runtime overhead and to remove some of the limitations listed in Section 4. Meanwhile, we hope to receive feedback from the community that will help us improve its usability.

Our ongoing research plan views liblog as a platform for building further analysis and failure detection tools. Specifically, replaying multiple processes together provides a convenient arena for analyzing distributed state. We see great potential for consistency checking and distributed predicate evaluation tools.

References

- [CGC⁺03] Frank Cornelis, Andy Georges, Mark Christiaens, Michiel Ronsse, Tom Ghesquiere, and Koen De Bosschere. A taxonomy of execution replay systems. In *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.
- [Ckp] Ckpt project website. <http://www.cs.wisc.edu/zandy/ckpt/>.
- [DFD96] Carl Dionne, Marc Feeley, and Jocelyn Desbiens. A taxonomy of distributed debuggers based on execution replay. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, CA, August 1996.
- [FLRS05] Michael J. Freedman, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-transitive connectivity and dhds. In *Proceedings of WORLDS*, December 2005.
- [GDB] Gnu debugger website. <http://gnu.org/software/gdb/>.
- [HH05] Alex Ho and Steven Hand. On the design of a pervasive debugger. In *Proceedings of the International Symposium on Automated Analysis-Driven Debugging*, September 2005.
- [Hus02] Joel Huselius. Debugging parallel systems: A state of the art report. Technical Report MDH-MRTC-63/2002-1-SE, Maelardalen Real-Time Research Centre, September 2002.
- [JJK⁺06] Dilip Joseph, Jayanthkumar Kannan, Ayumu Kubota, Karthik Lakshminarayanan, Ion Stoica, and Klaus Wehrle. Ocala: An architecture for supporting legacy applications over overlays. In *Proceedings of NSDI*, May 2006.
- [KDC05] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX 2005 Annual Technical Conference*, June 2005.
- [KSC00] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. Deterministic replay of distributed java applications. In *Proceedings of International Parallel and Distributed Processing Symposium*, May 2000.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [NM92] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of the International Conference on Supercomputing*, November 1992.
- [NPC05] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture*, 2005.
- [PL] Planet-lab project website. <http://planet-lab.org/>.
- [Pur] Purify website. <http://ibm.com/software/awdtools/purify/>.
- [Py] Python project website. <http://python.org/>.
- [RBdK99] Michiel Ronsse, Koenraad De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay for an mpi-based multi-threaded runtime system. In *Proceedings of the International Conference Parallel Computing*, 1999.
- [RZ97] Michiel Ronsse and Willy Zwaenepoel. Execution replay for treadmarks. In *Proceedings of EUROMICRO Workshop on Parallel and Distributed Processing*, January 1997.
- [Sai05] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the International Symposium on Automated Analysis-Driven Debugging*, September 2005.
- [SAZ⁺02] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM*, August 2002.
- [SH] Simics hindsight. <http://www.virtutech.com/products/simics-hindsight.html>.
- [SKAZ04] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX 2004 Annual Technical Conference*, June 2004.
- [SSL] Openssl project website. <http://openssl.org/>.
- [TH00] Henrik Thane and Hans Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of 12th Euromicro Conference on Real-Time Systems*, June 2000.
- [Val] Valgrind project website. <http://valgrind.org/>.
- [XBH03] Min Xu, Rastislav Bodik, and Mark Hill. A flight data recorder for enabling fullsystem multiprocessor deterministic replay. In *30th International Symposium on Computer Architecture*, 2003.

Loose Synchronization for Large-Scale Networked Systems

Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat

University of California, San Diego

{jalbrecht, ctuttle, snoeren, vahdat}@cs.ucsd.edu

Abstract

Traditionally, synchronization barriers ensure that no co-operating process advances beyond a specified point until all processes have reached that point. In heterogeneous large-scale distributed computing environments, with unreliable network links and machines that may become overloaded and unresponsive, traditional barrier semantics are too strict to be effective for a range of emerging applications. In this paper, we explore several relaxations, and introduce a *partial barrier*, a synchronization primitive designed to enhance liveness in loosely coupled networked systems. Partial barriers are robust to variable network conditions; rather than attempting to hide the asynchrony inherent to wide-area settings, they enable appropriate application-level responses. We evaluate the improved performance of partial barriers by integrating them into three publicly available distributed applications running across PlanetLab. Further, we show how partial barriers simplify a re-implementation of MapReduce that targets wide-area environments.

1 Introduction

Today's resource-hungry applications are increasingly deployed across large-scale networked systems, where significant variations in processor performance, network connectivity, and node reliability are the norm. These installations lie in stark contrast to the tightly-coupled cluster and supercomputer environments traditionally employed by compute-intensive applications. What remains unchanged, however, is the need to synchronize various phases of computation across the participating nodes. The realities of this new environment place additional demands on synchronization mechanisms; while existing techniques provide correct operation, performance is often severely degraded. We show that new, relaxed synchronization semantics can provide significant performance improvements in wide-area distributed systems.

Synchronizing parallel, distributed computation has long been the focus of significant research effort. At a high level, the goal is to ensure that concurrent computation tasks—across independent threads, processors, nodes in a cluster, or spread across the Internet—are able to make

independent forward progress while still maintaining some higher-level correctness semantic. Perhaps the simplest synchronization primitive is the barrier [19], which establishes a rendezvous point in the computation that all concurrent nodes must reach before any are allowed to continue. Bulk synchronous parallel programs running on MPPs or clusters of workstations employ barriers to perform computation and communication in phases, transitioning from one consistent view of shared underlying data structures to another.

In past work, barriers and other synchronization primitives have defined strict semantics that ensure safety—*i.e.*, that no node falls out of lock-step with the others—at the expense of liveness. In particular, if employed naively, a parallel computation using barrier synchronization moves forward at the pace of the slowest node and the entire computation must be aborted if any node fails. In closely-coupled supercomputer or cluster environments, skillful programmers optimize their applications to reduce these synchronization overheads by leveraging knowledge about the relative speed of individual nodes. Further, dataflow can be carefully crafted based upon an understanding of transfer times and access latencies to prevent competing demand for the I/O bus. Finally, recovering from failure is frequently expensive; thus, failure during computation is expected to be rare.

In large-scale networked systems—where individual node speeds are unknown and variable, communication topologies are unpredictable, and failure is commonplace—applications must be robust to a range of operating conditions. The performance tuning common in cluster and supercomputing environments is impractical, severely limiting the performance of synchronized concurrent applications. Further, individual node failures are almost inevitable, hence applications are generally engineered to adapt to or recover from failure. Our work focuses on distributed settings where online performance optimization is paramount and correctness is ensured through other mechanisms. We introduce adaptive mechanisms to control the degree of concurrency in cases where parallel execution appears to be degrading performance due to self-interference.

At a high level, emerging distributed applications all implement some form of traditional synchronizing barriers. While not necessarily executing a SIMD instruction stream, different instances of the distributed computation reach a shared point in the global computation. Relative to traditional barriers, however, a node arriving at a barrier need not necessarily block waiting for all other instances to arrive—doing so would likely sacrifice efficiency or even liveness as the system waits for either slow or failed nodes. Similarly, releasing a barrier does not necessarily imply that all nodes should pass through the barrier simultaneously—e.g., simultaneously releasing thousands of nodes to download a software package effectively mounts a denial-of-service attack against the target repository. Instead, applications manage the entry and release semantics of their logical barriers in an *ad hoc*, application-specific manner.

This paper defines, implements, and evaluates a new synchronization primitive that meets the requirements of a broad range of large-scale network services. In this context, we make the following contributions:

- We introduce a *partial barrier*, a synchronization primitive designed for heterogeneous failure-prone environments. By relaxing traditional semantics, partial barriers enhance liveness in the face of slow, failed, and self-interfering nodes.
- Based on the observation that the arrival rate at a barrier will often form a heavy-tailed distribution, we design a heuristic to dynamically detect the *knee of the curve*—the point at which arrivals slow considerably—allowing applications to continue despite slow nodes. We also adapt the rate of release from a barrier to prevent performance degradation in the face of self-interfering processes.
- We adapt three publicly available, wide-area services to use partial barriers for synchronization, and reimplement a fourth. In particular, we use partial barriers to reduce the implementation complexity of an Internet-scale version of MapReduce [9] running across PlanetLab. In all four cases, partial barriers result in a significant performance improvement.

2 Distributed barriers

We refer to nodes as *entering* a barrier when they reach a point in the computation that requires synchronization. When a barrier *releases* or *fires* (we use the terms interchangeably), the blocked nodes are allowed to proceed. According to strict barrier semantics, ensuring safety, *i.e.*, global synchronization, requires all nodes to reach a synchronization point before any node proceeds. In the face of wide variability in performance and prominent failures, however, strict enforcement may force the ma-

jority of nodes to block while waiting for a handful of slow or failed nodes to catch up. Many wide-area applications already have the ability to reconfigure themselves to tolerate node failure. We can harness this functionality to avoid excessive waits at barriers: once slow nodes are identified, they can be removed and possibly replaced by quicker nodes for the remainder of the execution.

One of the important questions, then, is determining when to release a barrier, even if all nodes have not arrived at the synchronization point. That is, it is important to dynamically determine the point where waiting for additional nodes to enter the barrier will cost the application more than the benefit brought by any additional arriving nodes in the future. This determination often depends on the semantics of individual applications. Even with full knowledge of application behavior, making the decision appropriately requires future knowledge. One contribution of this work is a technique to dynamically identify “knees” in the node arrival process, *i.e.*, points where the arrival process significantly slows. By communicating these points to the application we allow it to make informed decisions regarding when to release.

A primary contribution of this work is the definition of relaxed barrier semantics to support a variety of wide-area distributed computations. To motivate our proposed extensions consider the following application scenarios:

Application initialization. Many distributed applications require a startup phase to initialize their local state and to coordinate with remote participants before beginning their operation. Typically, developers introduce an artificial delay judged to be sufficient to allow the system to stabilize. If each node entered a barrier upon completing the initialization phase, developers would be freed from introducing arbitrarily chosen delays into the interactive development/debugging cycle.

Phased computation and communication. Scientific applications and large-scale computations often operate in phases consisting of one or many sequential, local computations, followed by periods of global communication or parallel computation. These computations naturally map to the barrier abstraction: one phase of the computation must complete before proceeding to the next phase. Other applications operate on a work queue that distributes tasks to available machines based on the rate that they complete individual tasks. Here, a barrier may serve as a natural point for distributing work.

Coordinated measurement. Many distributed applications measure network characteristics. However, uncoordinated measurements can self-interfere, leading to wasted effort and incorrect results. Such systems benefit from a mechanism that limits the number of nodes that simultaneously perform probes. Barriers are capable of

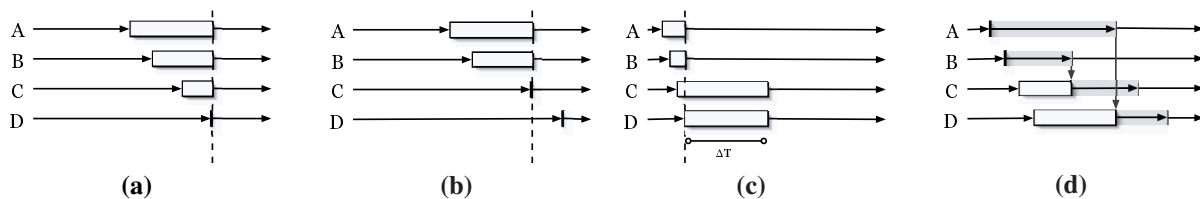


Figure 1: (a) Traditional semantics: All hosts enter the barrier (indicated by the white boxes) and are simultaneously released (indicated by dotted line). (b) Early entry: The barrier fires after 75% of the hosts arrive. (c) Throttled release: Hosts are released in pairs every ΔT seconds. (d) Counting semaphore: No more than 2 processes are simultaneously allowed into a “critical section” (indicated by the grey regions). When one node exits the critical section, another host is allowed to enter.

providing this needed functionality. In this case, the application uses two barriers to delimit a “critical section” of code (e.g., a network measurement). The first barrier releases some maximum number of nodes into the critical section at a time and waits until these nodes reach the second barrier, thereby exiting the critical section, before releasing the next round.

To further clarify the goal of our work, it is perhaps worthwhile to consider what we *are not* trying to accomplish. Partial barriers provide only a loose form of group membership [8, 16, 27]. In particular, partial barriers do not provide any ordering of messages relative to barriers as in virtual synchrony [2, 3], nor do partial barriers require that all participants come to consensus regarding a view change [24]. In effect, we strive to construct an abstraction that exposes the asynchrony and the failures prevalent in wide-area computing environments in a manner that allows the application to make dynamic and adaptive decisions as to how to respond.

It is also important to realize that not all applications will benefit from relaxed synchronization semantics. The correctness of certain classes of applications cannot be guaranteed without complete synchronization. For example, some applications may require a minimum number of hosts to complete a computation. Other applications may approximate a measurement (such as network delay) and continuing without all nodes reduces the accuracy of the result. However, many distributed applications, including the ones described in Section 5, can afford to sacrifice global synchronization without negatively impacting the results. These applications either support dynamically degrading the computation, or are robust to failures and can tolerate mid-computation reconfigurations. Our results indicate that for applications that are willing and able to sacrifice safety, our semantics have the potential to improve performance significantly.

3 Design and implementation

Partial barriers are a set of semantic extensions to the traditional barrier abstraction. Our implementation has a simple interface for customizing barrier functionality.

This section describes these extended semantics, details our API, and presents the implementation details.

3.1 Design

We define two new barrier semantics to support emerging wide-area computing paradigms and discuss each below.

Early entry – Traditional barriers require all nodes to enter a barrier before any node may pass through, as in Figure 1(a). A partial barrier with early entry is instantiated with a timeout, a minimum percentage of entering nodes, or both. Once the barrier has met either of the specified preconditions, nodes that have already entered a barrier are allowed to pass through without waiting for the remaining slow nodes (Figure 1(b)). Alternatively, an application may instead choose to receive callbacks as nodes enter and manually release the barrier, enabling the evaluation of arbitrary predicates.

Throttled-release – Typically, a barrier releases all nodes simultaneously when a barrier’s precondition is met. A partial barrier with throttled-release specifies a rate of release, such as two nodes every ΔT seconds as shown in Figure 1(c). A special variation of throttled-release barriers allows applications to limit the number of nodes that simultaneously exist in a “critical section” of code, creating an instance of a counting semaphore [10] (shown in Figure 1(d)), which may be used, for example, to throttle the number of nodes that simultaneously perform network measurements or software downloads. A critical distinction between traditional counting semaphores and partial barriers, however, is our support for failure. For instance, if a sufficient number of slow or soon-to-fail nodes pass a counting semaphore, they will limit access to other participants, possibly forever. Thus, as with early-entry barriers, throttled-release barriers eventually time out slow or failed nodes, allowing the system as a whole to make forward progress despite individual failures.

One issue that our proposed semantics introduce that does not arise with strict barrier semantics is handling nodes performing late entry, *i.e.*, arriving at an already released barrier. We support two options to address this

```

class Barrier {
    Barrier(string name, int max, int timeout,
           int percent, int minWait);
    static void setManager(string Hostname);
    void enter(string label, string Hostname);
    void setEnterCallback(bool (*callbackFunc)(string label,
        string Hostname, bool default), int timeout);
    map<string label, string Hostname> getHosts(void);
}

```

Figure 2: Barrier instantiation API.

case: i) pass-through semantics that allow the node to proceed with the next phase of the computation even though it arrived late; ii) catch-up semantics that issue an exception allowing the application to reintegrate the node into the mainline computation in an application-specific manner, which may involve skipping ahead to the next barrier (omitting the intervening section of code) in an effort to catch up to the other nodes.

3.2 Partial barrier API

Figure 2 summarizes the partial barrier API from an application's perspective. Each participating node initializes a barrier with a constructor that takes the following arguments: `name`, `max`, `timeout`, `percent`, and `minWait`. `name` is a globally unique identifier for the barrier. `max` specifies the maximum number of participants in the barrier. (While we do not require *a priori* knowledge of participant identity, it would be straightforward to add.) The `timeout` in milliseconds sets the maximum time that can pass from the point where the first node enters a barrier before the barrier is released. The `percent` field similarly captures a minimum percentage out of the maximum number nodes that must reach the barrier to activate early release. The `minWait` field is associated with the `percent` field and specifies a minimum amount of time to wait (even if the specified percentage of nodes have reached) before releasing the barrier with less than the maximum number of nodes. Without this field, the barrier will deterministically be released upon reaching the threshold percent of entering nodes even when all nodes are entering rapidly. However, the barrier is always released if `max` nodes arrive, regardless of `minWait`. The `timeout` field overrides the `percent` and `minWait` fields; the barrier will fire if the timeout is reached, regardless of the percentage of nodes entering the barrier. The last three parameters to the constructor are optional—if left unspecified the barrier will operate as a traditional synchronization barrier.

Coordination of barrier participants is controlled by a barrier manager whose identity is specified at startup by the `setManager()` method. Participants call the barrier's `enter()` method and pass in their `Hostname`

and `label` when they reach the appropriate point in their execution. (The `label` argument supports advanced functionality such as load balancing for MapReduce as described in Section 5.4.) The participant's `enter()` method notifies the manager that the particular node has reached the synchronization point. Our implementation supports blocking calls to `enter()` (as described here) or optionally a callback-based mechanism where the entering node is free to perform other functionality until the appropriate callback is received.

While our standard API provides simplistic support for the early release of a barrier, an application may maintain its own state to determine when a particular barrier should fire and to manage any side effects associated with barrier entry or release. For instance, a barrier manager may wish to kill processes arriving late to a particular (already released) barrier. To support application-specific functionality, the `setEnterCallback()` method specifies a function to be called when any node enters a barrier. The callback takes the `label` and `Hostname` passed to the `enter()` method and a boolean variable that specifies whether the manager would normally release the barrier upon this entry. The callback function returns a boolean value to specify whether the barrier should actually be released or not, potentially overriding the manager's decision. A second argument to `setEnterCallback()` called `timeout` specifies a maximum amount of time that may pass before successive invocations of the callback. This prevents the situation where the application waits a potentially infinite amount of time for the next node to arrive before deciding to release the barrier. We use this callback API to implement our adaptive release techniques presented in Section 4.

Barrier participants may wish to learn the identity of all hosts that passed through a barrier, similar to (but with relaxed semantics from) view advancement or GBCAST in virtual synchrony [4]. The `getHosts()` method returns a map of `Hostnames` and `labels` through a remote procedure call with the barrier manager. If many hosts are interested in membership information, it can optionally be propagated from the barrier manager to all nodes by default as part of the barrier release operation.

Figure 3 describes a subclass of `Barrier`, called `ThrottleBarrier`, with *throttled-release* semantics. These semantics allow for a pre-determined subset of the maximum number of nodes to be released at a specified rate. The methods `setThrottleReleasePercent()` and `setThrottleReleaseCount()` periodically release a percentage and number of nodes, respectively, once the barrier fires. `setThrottleReleaseTimeout()` specifies the periodicity of release.

```

class ThrottleBarrier extends Barrier {
    void setThrottleReleasePercent(int percent);
    void setThrottleReleaseCount(int count);
    void setThrottleReleaseTimeout(int timeout);
}
class SemaphoreBarrier extends Barrier {
    void setSemaphoreCount(int count);
    void setSemaphoreTimeout(int timeout);
    void release(string label, string Hostname);
    void setReleaseCallback(int (*callbackFunc)(string label,
        string Hostname, int default), int timeout);
}

```

Figure 3: ThrottleBarrier and SemaphoreBarrier API.

Figure 3 also details a variant of throttled-release barriers, `SemaphoreBarrier`, which specifies a maximum number of nodes that may simultaneously enter a critical section. A `SemaphoreBarrier` extends the throttled-release semantics further by placing a barrier at the beginning and end of a critical section of activity to ensure that only a specific number of nodes pass into the critical section simultaneously. One key difference for this type of barrier is that it does not require any minimum number of nodes to enter the barrier before beginning to release nodes into the subsequent critical section. It simply mandates a maximum number of nodes that may simultaneously enter the critical section. The `setSemaphoreCount()` method sets this maximum number. Nodes call the barrier's `release()` method upon completing the subsequent critical section, allowing the barrier to release additional nodes. `setSemaphoreTimeout()` allows for timing out nodes that enter the critical section but do not complete within a maximum amount of time. In this case, they are assumed to have failed, enabling the release of additional nodes. The `setReleaseCallback()` enables application-specific release policies and timeout of slow or failed nodes in the critical section. The callback function in `setReleaseCallback()` returns the number of hosts to be released.

3.3 Implementation

Partial barrier participants implement the interface described above while a separate barrier manager coordinates communication across nodes. Our implementation of partial barriers consists of approximately 3,000 lines of C++ code. At a high level, a node calling `enter()` transmits a `BARRIER_REACHED` message using TCP to the manager with the calling host's unique identifier, barrier name, and label. The manager updates its local state for the barrier, including the set of nodes that have thus far entered the barrier, and performs appropriate callbacks as necessary. The manager starts a timer to support various release semantics if this is the first node

entering the barrier and subsequently records the inter-arrival times between nodes entering the barrier.

If a sufficient number of nodes enter the barrier or a specified amount of time passes, the manager transmits `FIRE` messages using TCP to all nodes that have entered the barrier. For throttled release barriers, the manager releases the specified number of nodes from the barrier in FIFO order. The manager also sets a timer as specified by `setThrottleReleaseTimeout()` to release additional nodes from the barrier when appropriate.

For semaphore barriers, the manager releases the number of nodes specified by `setSemaphoreCount()` and, if specified by `setSemaphoreTimeout()`, also sets a timer to expire for each node entering the critical section. Each call to `enter()` transmits a `SEMAPHORE_REACHED` message to the manager. In response, the manager starts the timer associated with the calling node. If the semaphore timer associated with the node expires before receiving the corresponding `SEMAPHORE_RELEASED` message, the manager assumes that node has either failed or is proceeding sufficiently slowly that an additional node should be released into the critical section. Each `SEMAPHORE_RELEASED` message releases one additional node into the critical section.

For all barriers, the manager must gracefully handle nodes arriving late, *i.e.*, after the barrier has fired. We employ two techniques to address this case. For pass-through semantics, the manager transmits a `LATE_FIRE` message to the calling node, releasing it from the barrier. In catch-up semantics, the manager issues an exception and transmits a `CATCH_UP` message to the node. Catch-up semantics allow applications to respond to the exception and determine how to reintegrate the node into the computation in an application-specific manner.

The type of barrier—pass-through or catch-up—is specified at barrier creation time (intentionally not shown in Figure 2 for clarity). Nodes calling `enter()` may register local callbacks with the arrival of either `LATE_FIRE` or `CATCH_UP` messages for application-specific re-synchronization with the mainline computation, or perhaps to exit the local computation altogether if re-synchronization is not possible.

3.4 Fault tolerance

One concern with our centralized barrier manager is tolerating manager faults. We improve overall system robustness with support for replicated managers. Our algorithm is a variant of traditional primary/backup systems: each participant maintains an ordered list of barrier controllers. Any message sent from a client to the logical barrier manager is sent to all controllers on the list. Because application-specific entry callbacks may be

non-deterministic, a straightforward replicated state machine approach where each barrier controller simultaneously decides when to fire is insufficient. Instead, the primary controller forwards all `BARRIER_REACHED` messages to the backup controllers. These messages act as implicit “keep alive” messages from the primary. If a backup controller receives `BARRIER_REACHED` messages from clients but not the primary for a sufficient period of time, the first backup determines the primary has failed and assumes the role of primary controller. The secondary backup takes over should the primary fail, and so on. Note that our approach admits the case where multiple controllers simultaneously act as the primary controller for a short period of time. Clients ignore duplicate `FIRE` messages for the same barrier, so progress is assured, and one controller eventually emerges as primary.

Although using the replicated manager scheme described above lowers the probability of losing `BARRIER_REACHED` messages, it does not provide any increased reliability with respect to messages sent from the manager(s) to the remote hosts. All messages are sent using reliable TCP connections. If a connection fails between the manager and a remote host, however, messages may be lost. For example, suppose the TCP connections between the manager and some subset of the remote hosts break just after the manager sends a `FIRE` message to all participants. Similarly, if a group of nodes fails after entering the barrier, but before receiving the `FIRE` message, the failure may go undetected until after the controller transmits the `FIRE` messages. In these cases, the manager will attempt to send `FIRE` messages to all participants, and detect the TCP failures after the connections time out. Such ambiguity is unavoidable in asynchronous systems; the manager simply informs the application of the failure(s) via a callback and lets the application decide the appropriate recovery action. As with any other failure, the application may choose to continue execution and ignore the failures, attempt to find new hosts to replace the failed ones, or to even abort the execution entirely.

4 Adaptive release

Unfortunately, our extended barrier semantics introduce additional parameters: the threshold for early release and the concurrency level in throttled release. Experience has shown it is often difficult to select values that are appropriate across heterogeneous and changing network conditions. Hence, we provide adaptive mechanisms to dynamically determine appropriate values.

4.1 Early release

There is a fundamental tradeoff in specifying an early-release threshold. If the threshold is too large, the application will wait unnecessarily for a relatively modest

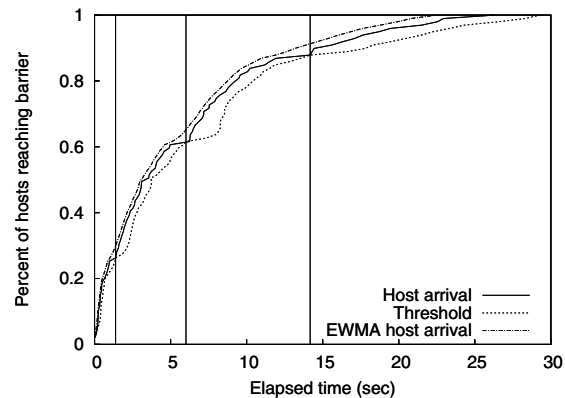


Figure 4: Dynamically determining the knee of arriving processes. Vertical bars indicate a knee detection.

number of additional nodes to enter the barrier; if too small, the application will lose the opportunity to have participation from other nodes had it just waited a bit longer. Thus, we use the barrier’s callback mechanism to determine release points in response to varying network conditions and node performance.

In our experience, the distribution of nodes’ arrivals at a barrier is often heavy-tailed: a relatively large portion of nodes arrive at the barrier quickly with a long tail of stragglers entering late. In this case, many of our target applications would wish to dynamically determine the “knee” of a particular arrival process and release the barrier upon reaching it. Unfortunately, while it can be straightforward to manually determine the knee off-line once all of the data for an arrival process is available, it is difficult to determine this point on-line.

Our heuristic, inspired by TCP retransmission timers and MONET [1], maintains an exponentially weighted moving average (EWMA) of the host arrival times (arr), and another EWMA of the deviation from this average for each measurement ($arrvar$). As each host arrives at the barrier, we record the arrival time of the host, as well as the deviation from the average. Then we recompute the EWMA for both arr and $arrvar$, and use the values to compute a maximum wait threshold of $arr + 4 * arrvar$. This threshold indicates the maximum time we are willing to wait for the next host to arrive before firing the barrier. If the next host does not arrive at the barrier before the maximum wait threshold passes, we assume that a knee has been reached. Figure 4 illustrates how these values interact for a simulated group of 100 hosts entering a barrier with randomly generated exponential inter-arrival times. Notice that a knee occurs each time the host arrival time intersects the threshold line.

With the capability to detect multiple knees, it is important to provide a way for applications to pick the right knee and avoid firing earlier or later than desired. Ag-

gressive applications may choose to fire the barrier when the first knee is detected. Conservative applications may wish to wait until some specified amount of time has passed, or a minimum percentage of hosts have entered the barrier before firing. To support both aggressive and conservative applications, our algorithm allows the application to specify a minimum percentage of hosts, minimum waiting time, or both for each barrier. If an application specifies a minimum waiting time of 5 seconds, knees detected before 5 seconds are ignored. Similarly, if a minimum host percentage of 50% is specified, the knee detector ignores knees detected before 50% of the total hosts have entered the barrier. If both values are specified, the knee detector uses the more conservative threshold so that both requirements (time and host percentage) are met before firing.

One variation in our approach compared to other related approaches is the values for the weights in the moving averages. In the RFC for computing TCP retransmission timers [7], the weight in the EWMA of the *rtt* places a heavier weight (0.875) on previous delay measurements. This value works well for TCP since the average delay is expected to remain relatively constant over time. In our case, however, we expect the average arrival time to increase, and thus we decreased the weight to be 0.70 for previous measurements of *arr*. This allows our *arr* value to more closely follow the actual data being recorded. When measuring the average deviation, which is computed by averaging $|sample - arr|$ (where *sample* represents the latest arrival time recorded), we used a weight of 0.75 for previous measurements, which is the same weight used in TCP for the variation in *rtt*.

4.2 Throttled release

We also employ an adaptive method to dynamically adjust the amount of concurrency in the “critical section” of a semaphore barrier. In many applications, it is impractical to select a single value which performs well under all conditions. Similar in spirit to SEDA’s thread-pool controller [35], our adaptive release algorithm selects an appropriate concurrency level based upon recent release times. The algorithm starts with low level of concurrency and increases the degree of concurrency until response times worsen; it then backs off and repeats, oscillating about the optimum value.

Mathematically, the algorithm compares the median of the distributions of recent and overall release times. For example, if there are 15 hosts in the critical section when the 45th host is released, the algorithm computes the median release time of the last 15 releases, and of all 45. If the latest median is more than 50% greater than the overall median, no additional hosts are released, thus reducing the level of concurrency to 14 hosts. If the latest

median is more than 10% but less than 50% greater than the overall median, one host is released, maintaining a level of concurrency of 15. In all other cases, two hosts are released, increasing the concurrency level to 16. The thresholds and differences in size are selected to increase the degree of concurrency whenever possible, but keep the magnitude of each change small.

5 Applications

We integrated partial barriers into three wide-area, distributed applications with a range of synchronization requirements. We reimplemented a fourth application whose source code was unavailable. While a detailed discussion of these applications is beyond the scope of this paper, we present a brief overview to facilitate understanding of our performance evaluation in Section 6.

5.1 Plush

Plush [29] is a tool for configuring and managing large-scale distributed applications, such as PlanetLab [28] and the Grid [13]. Users specify a description of: i) a set of nodes to run their application on; ii) the set of software packages, including the application itself and any necessary data files, that should be installed on each node; and iii) a directed acyclic graph of individual processes that should be run, in order, on each node.

Plush may be used to manage long-running services or interactive experimental evaluations of distributed applications. In the latter case, developers typically configure a set of machines (perhaps installing the latest version of the application binary) and then start the processes at approximately the same time on all participating nodes. A barrier may be naturally inserted between the tasks of node configuration and process startup. However, in the heterogeneous PlanetLab environment, the time to configure a set of nodes with the requisite software can vary widely or fail entirely at individual hosts. In this case, it is often beneficial to timeout the “software configuration/install” barrier and either proceed with the available nodes or to recruit additional nodes.

5.2 Bullet

Bullet [21] is an overlay-based large-file distribution infrastructure. In Bullet, a source transmits a file to receivers spread across the Internet. As part of the bootstrap process, all receivers join the overlay by initially contacting the source before settling on their final position in the overlay. The published quantitative evaluation of Bullet presents a number of experiments across PlanetLab. However, to make performance results experimentally meaningful when measuring behavior across a large number of receivers, the authors hard-coded a 30-second delay at the sender from the time that it starts

to the time that it begins data transmission. While typically sufficient for the particular targeted configuration, the timeout would often be too long, unnecessarily extending turnaround time for experimentation and interactive debugging. Depending on overall system load and number of participants, the timeout would also sometimes be too short, meaning that some nodes would not complete the join process at the time the sender begins transmitting. While this latter case is not a problem for the correct behavior of the system, it makes interpreting experimental results difficult.

To address this limitation, we integrated partial barriers into the Bullet implementation. This integration was straightforward. Once the join process completes, each node simply enters a barrier. The Bullet sender registers for a callback with the barrier manager to be notified of a barrier release, at which point it begins transmitting data. By calling the `getHosts()` method, the sender can record the identities of nodes that should be considered in interpreting the experimental results. The barrier manager notes the identity of nodes entering the barrier late and instructs them to exit rather than proceed with retrieving the file.

5.3 EMAN

EMAN [12] is a publicly available software package used for reconstructing 3D models of particles using 2D electron micrographs. The program takes a 2D micrograph image as input and then repeatedly runs a “refinement” process on the data to create a 3D model. Each iteration of the refinement consists of both computationally inexpensive sequential computations and computationally expensive parallel computations.

Barriers separate sequential and parallel stages of computation in EMAN. Using partial barrier semantics adds the benefit of being able to detect slow nodes, allowing the application to redistribute tasks to faster machines during the parallel phase of computation. In addition to slow processors, the knee detector also detects machines with low bandwidth capacities and reallocates their work to machines with higher bandwidth. In our test, each refinement phase requires approximately 240 MB of data to be transferred to each node, and machines with low bandwidth links have a significant impact on the overall completion time if their work is not reallocated to different machines. Since the sequential phases run on a single machine, partial barriers are most applicable to the parallel phases of computation. We use the publicly available version of EMAN for our experiments. We wrote a 50-line Perl script to run the parallel phase of computation on 98 PlanetLab machines using Plush.

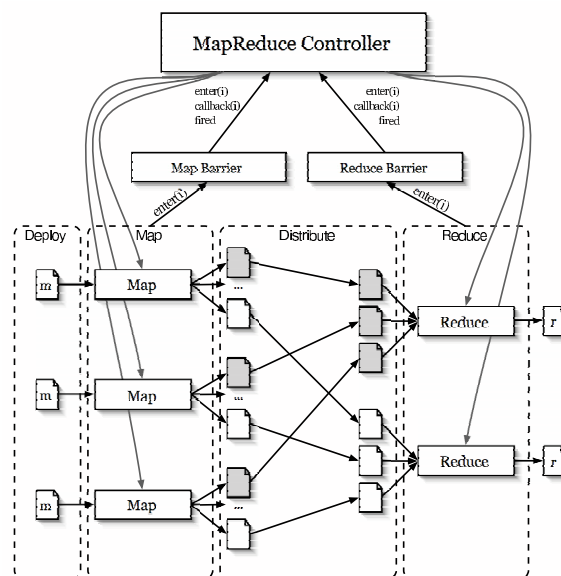


Figure 5: MapReduce execution. As each map task completes, `enter(i)` is called with the unique task identifier. Once all m tasks have entered the Map barrier, it is released. When the MapReduce controller is notified that the Map barrier has fired, the r reduce tasks are distributed and begin execution. When all r reduce tasks have entered the Reduce barrier, MapReduce is complete. In both barriers, `callback(i)` informs the MapReduce controller of task completions.

5.4 MapReduce

MapReduce [9] is a toolkit for application-specific parallel processing of large volumes of data. The model involves partitioning the input data into smaller *splits* of data, and spreading them across a cluster of worker nodes. Each worker node applies a *map* function to the splits of data that they receive, producing intermediate key/value pairs that are periodically written to specific locations on disk. The MapReduce master node tracks these locations, and eventually notifies another set of worker nodes that intermediate data is ready. This second set of workers aggregate the data and pass it to the *reduce* function. This function processes the data to produce a final output file.

Our implementation of MapReduce leverages partial barriers to manage phases of the computation and to orchestrate the flow of data among nodes across the wide area. In our design, we have m map tasks and corresponding input files, n total nodes hosting the computation, and r reduce tasks. A central MapReduce controller distributes the m split input files to a set of available nodes (our current implementation runs on PlanetLab), and spawns the

map process on each node. When the map tasks finish, intermediate files are written back to a central repository, and then redistributed to r hosts, who eventually execute the r reduce tasks. There are a number of natural barriers in this application, as shown in Figure 5, corresponding to completion of: i) the initial distribution of m split files to appropriate nodes; ii) executing m map functions; iii) the redistribution of the intermediate files to appropriate nodes, and iv) executing r reduce functions.

As with the original MapReduce work, the load balancing aspects corresponding to barriers (ii) and (iv) (from the previous paragraph) are of particular interest. Recall that although there are m map tasks, the same physical host may execute multiple map tasks. In these cases, the goal is not necessarily to wait for all n hosts to reach the barrier, but for all m or r logical tasks to complete. Thus, we extended the original barrier entry semantics described in Section 3 to support synchronizing barriers at the level of a set of logical, uniquely named tasks or processes, rather than a set of physical hosts. To support this, we simply invoke the `enter()` method of the barrier API (see Figure 2) upon completing a particular map or reduce function. In addition to the physical hostname, we send a label corresponding to a globally unique name for the particular map or reduce task. Thus, rather than waiting for n hosts, the barrier instead waits for m or r unique labels to enter the barrier before firing.

For a sufficiently large and heterogeneous distributed system, performance at individual nodes varies widely. Such variability often results in a heavy-tailed distribution for the completion of individual tasks, meaning that while most tasks will complete quickly, the overall time will be dominated by the performance of the slow nodes. The original MapReduce work noted that one of the common problems experienced during execution is the presence of “straggler” nodes that take an unusually long time to complete a map or reduce task. Although the authors mentioned an application-specific solution to this problem, by using partial barriers in our implementation we were able to provide a more general solution that achieved the same results. We use the arrival rate of map/reduce tasks at their respective barriers to respawn a subset of the tasks that were proceeding slowly.

By using the knee detector described in Section 4, we are able to dynamically determine the transition point between rapid arrivals and the long tail of stragglers. However, rather than releasing the barrier at this point, the MapReduce controller receives a callback from the barrier manager, and instead performs load rebalancing functionality by spawning additional copies of outstanding tasks on nodes disjoint from the ones hosting the slower tasks (potentially first distributing the necessary input/intermediate files). As in the original implemen-

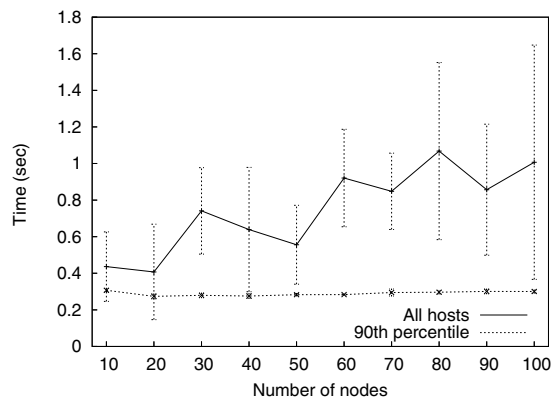


Figure 6: Scalability of centralized barrier implementation. “All hosts” line shows the average time across 5 runs for barrier manager to receive `BARRIER_REACHED` messages from all hosts. “90th percentile” line shows the average time across 5 runs for barrier manager to receive `BARRIER_REACHED` messages from 90% of all hosts.

tation of MapReduce, the barrier is not concerned with what copies of the computation complete first; the goal is for all m or r tasks to complete as quickly as possible.

The completion rate of tasks also provides an estimate of the throughput available at individual nodes, influencing future task placement decisions. Nodes with high throughput are assigned more tasks in the future. Our performance evaluation in Section 6 quantifies the benefits of this approach. Cluster-based MapReduce [9] also found this rebalancing to be critical for improving performance in more tightly controlled cluster settings, but did not describe a precise approach for determining when to spawn additional instances of a given computation.

6 Evaluation

The goal of this section is to quantify the utility of partial barriers for a range of application scenarios. For all experiments, we randomly chose subsets from a pool of 130 responsive PlanetLab nodes.

6.1 Scalability

To estimate baseline barrier scalability, we measure the time it takes to move between two barriers for an increasing number of hosts. In this experiment, the controller waits for all hosts to reach the first barrier. All hosts are released, and then immediately enter the second barrier. We measure the time between when the barrier manager sends the first `FIRE` message for the first barrier and receives the last `BARRIER_REACHED` message for the second barrier. No partial barrier semantics are used for these measurements. Figure 6 shows the average com-

pletion time for varying numbers of nodes across a total of five runs for each data point with standard deviation.

Notice that even for 100 nodes, the average time for the barrier manager to receive the last `BARRIER_REACHED` message for the second barrier is approximately 1 second. The large standard deviation values indicate that there is much variability in our results. This is due to the presence of straggler nodes that delay the firing for several seconds or more. The 90th percentile, on the other hand, has little variation and is relatively constant as the number of participants increases. This augurs well for the potential of partial barrier semantics to improve performance in the wide area. Overall, we are satisfied with the performance of our centralized barriers for 100 nodes; we expect to use hierarchy to scale significantly further.

6.2 Admission control

Next, we consider the benefits of a semaphore barrier to perform admission control for parallel software installation in Plush. Plush configures a set of wide-area hosts to execute a particular application. This process often involves installing the same software packages located on a central server. Simultaneously performing this install across hundreds of nodes can lead to thrashing at the server hosting the packages. The overall goal is to ensure sufficient parallelism such that the server is saturated (without thrashing) while balancing the average time to complete the download across all participants.

For our results, we use Plush to install the same 10-MB file on 100 PlanetLab hosts while varying the number of simultaneous downloads using a semaphore barrier. Figure 7 shows the results of this experiment. The data indicates that limiting parallelism can improve overall completion rate. Releasing too few hosts does not fully consume server resources, while releasing too many taxes available resources, increasing the time to first completion. This is evident in the graph since 25 simultaneous downloads finishes more quickly than both 100 and 10 simultaneous transfers.

While statically defining the number of hosts allowed to perform simultaneous downloads works well for our file transfer experiment, varying network conditions means that statically picking any single value is unlikely to perform well under all conditions. Some applications may benefit from a more dynamic throttled release technique that attempts to find the optimal number of hosts that maximizes throughput from the server without causing saturation. The “Adaptive Simultaneous Transfers” line in Figure 7 shows the performance of our adaptive release technique. In this example, the initial concurrency level is 15, and the level varies according to the duration of each transfer. In this experiment the adaptive al-

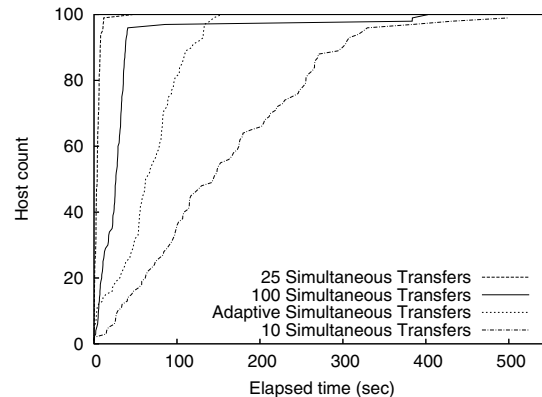


Figure 7: Software transfer from a high-speed server to PlanetLab hosts using a SemaphoreBarrier to limit the number of simultaneous file transfers.

gorithm line reaches 100% before the lines representing a fixed concurrency level of 10 or 100, but the algorithm was too conservative to match the optimal static level of 25 given the network conditions at the time.

6.3 Detecting knees

In this section we consider our ability to dynamically detect knees in a heavy-tailed arrival processes. We observe that when choosing a substantial number of time-shared PlanetLab hosts to perform the same amount of work, the completion time varies widely, often following an exponential distribution. This variation makes it difficult to coordinate distributed computation. To quantify our ability to more adaptively set timeouts and coordinate the behavior of multiple wide-area nodes, we used a barrier to synchronize senders and receivers in Bullet while running across a varying number of PlanetLab nodes. We set the barrier to dynamically detect the knee in the arrival process as described in Section 4. Upon reaching the knee, nodes already in the barrier are released; one side effect is that the sender begins data transmission. Bullet ignores all late arriving nodes.

Figure 8 plots the cumulative distribution of receivers that enter the startup barrier on the y -axis as a function of time progressing on the x -axis. Each curve corresponds to an experiment with 50, 90, or 130 PlanetLab receivers in the initial target set. The goal is to run with as many receivers as possible from the given initial set without waiting an undue amount of time for a small number of stragglers to complete startup. Interestingly, it is insufficient to filter for any static set of known “slow” nodes as performance tends to vary on fairly small time scales and can be influenced by multiple factors at a particular node (including available CPU, memory, and changing network conditions). Thus, manually choosing an appro-

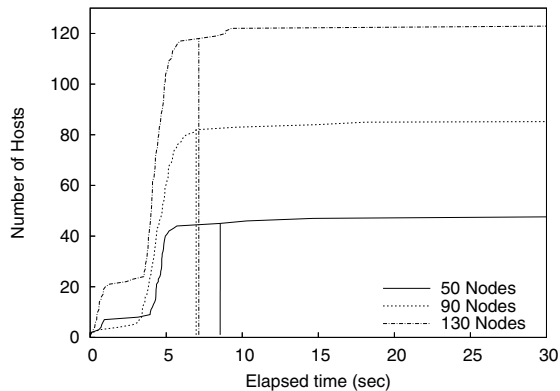


Figure 8: A startup barrier that regulates nodes joining a large-scale overlay. Vertical bars indicate when the barrier detects a knee and fires.

prate static set may be sufficient for one particular batch of runs but not likely the next.

Vertical lines in Figure 8 indicate where the barrier manager detects a knee and releases the barrier. Although we ran the experiment multiple times, for clarity we plot the results from a single run. While differences in time of day or initial node characteristics affect the quantitative behavior, the general shape of the curve is maintained. However, in all of our experiments, we are satisfied with our ability to dynamically determine the knee of the arrival process. The experiments are typically able to proceed with 85-90% of the initial set participating.

6.4 EMAN

For our next evaluation, we added a partial barrier to the parallel computation of EMAN’s refinement process. Upon detecting a knee, we reallocate tasks to faster machines. Figure 9 shows the results of running EMAN with and without partial barrier semantics. In this experiment we ran EMAN on the 98 most responsive PlanetLab machines. The workflow consists of several serial tasks (not shown), and a 98-way image classification step, run in parallel. Each participant first downloads a 40-MB archive containing the EMAN executables and a wrapper script. After unpacking the archive, each node downloads a unique 200-MB data file and begins running the classification process. At the end of the computation, each node generates 77 output files stored on the local disk, which EMAN merges into 77 “master” files once all tasks complete.

After 801 seconds, the barrier detects a knee and reallocates the remaining tasks to idle machines among those initially allocated to the experiment; these finish shortly afterward. The “knee” in the curve at approximately 300 seconds indicates that around 21 hosts have good connectivity to the data repository, while the rest have longer

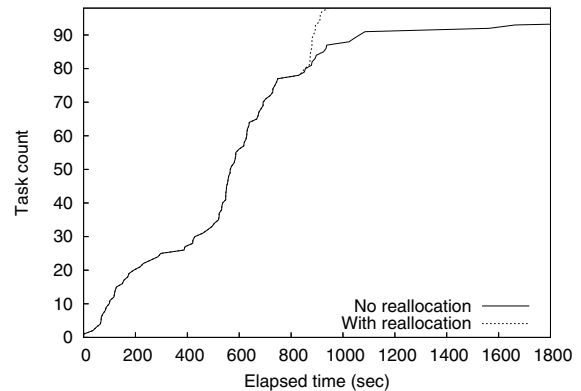


Figure 9: EMAN. Knee detected at 801 seconds. Total runtime is over 2700 seconds.

transfer times. While the knee detection algorithm detects a knee at 300 seconds, a minimum threshold of 60% prevents reconfiguration. The second knee is detected at 801 seconds, which starts a reconfiguration of 10 tasks. These tasks complete by 900 seconds, while the original set continue past 2700 seconds, for an overall speedup of more than 3.

6.5 MapReduce

We now consider an alternative use of partial barriers: to assist not with the synchronization of physical hosts or processors, but with load balancing of logical tasks spread across cooperating wide-area nodes. Further, we wish to determine whether we can dynamically detect knees in the arrival rate of individual tasks, spawning additional copies of tasks that appear to be proceeding slowly on a subset of nodes.

We conduct these experiments with our implementation of MapReduce (see Section 5.4) with $m = 480$ map tasks and $r = 30$ reduce tasks running across $n = 30$ PlanetLab hosts. During each of the map and reduce rounds, the MapReduce controller evenly partitions the tasks over the available hosts and starts the tasks asynchronously. For this experiment, each map task simply reads 2000 random words from a local file and counts the number of instances of certain words. This count is written to an intermediate output file based on the hash of the words. The task is CPU-bound and requires approximately 7 seconds to complete on an unloaded PlanetLab-class machine. The reduce tasks summarize these intermediate files with the same hash values.

Thus, each map and reduce task performs an approximately equal amount of work as in the original MapReduce work [9], though it would be useful to generalize to variable-length computation. When complete, a map or reduce task enters the associated barrier with a unique identifier for the completed work. The barrier manager

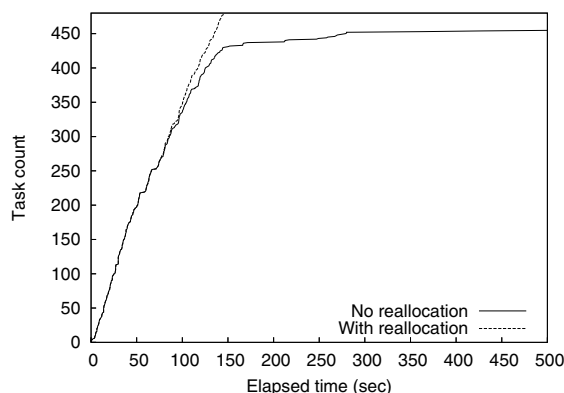


Figure 10: MapReduce: $m = 480$, $r = 30$, $n = 30$ with uniform prepartitioning of the data. Knee detection occurs at 68 seconds and callbacks enable rebalancing.

monitors the arrival rate and dynamically determines the knee, which is the where the completion rate begins to slow. We empirically determined that this slowing results from a handful of nodes that proceed substantially slower than the rest of the system. (Note that this phenomenon is not restricted to our wide-area environment; Dean and Ghemawat observed the same behavior for their runs on tightly coupled and more homogeneous clusters [9].) Thus, upon detecting the knee a callback to the MapReduce controller indicates that additional copies of the slow tasks should be respawned, ideally on nodes with the smallest number of outstanding tasks. Typically, by the time the knee is detected there are a number of hosts that have completed their initial allocation of work.

Figure 10 shows the performance of one MapReduce run both with and without task respawn upon detecting the knee. Figure 10 plots the cumulative number of completed tasks on the y -axis as a function of time progressing on the x -axis. We see that the load balancing enabled by barrier synchronization on abstract tasks is critical to overall system performance. With task respawn using knee detection, the barrier manager detects the knee approximately 68 seconds into the experiment after approximately 53% of the tasks have completed. At this point, the MapReduce controller (via a callback from the Barrier class) repartitions the remaining 47% of the tasks across available wide-area nodes. This is where the curves significantly diverge in the graph. Without dynamic rebalancing the completion rate transitions to a long-tail lasting more than 2500 seconds (though the graph only shows the first 500 seconds), while the completion rate largely maintains its initial slope when rebalancing is enabled. Overall, our barrier-based rebalancing results in a factor of sixteen speedup in completion time compared to proceeding with the initial mapping. Multiple additional runs showed similar results.

Note that this load balancing approach differs from the alternate approach of trying to predict the set of nodes likely to deliver the highest level of performance *a priori*. Unfortunately, predicting the performance of tasks with complex resource requirements on a shared computing infrastructure with dynamically varying CPU, network, and I/O characteristics is challenging in the best case and potentially intractable. We advocate a simpler approach that does not attempt to predict performance characteristics *a priori*. Rather, we simply choose nodes likely to perform well and empirically observe the utility of our decisions. Of course, this approach may only be appropriate for a particular class of distributed applications and comes at the cost of performing more work in absolute terms because certain computations are repeated. For the case depicted in Figure 10, approximately 30% of the work is repeated if we assume that the work on both the fast and slow nodes are run to completion (a pessimistic assumption as it is typically easy to kill tasks running on slow nodes once the fast instances complete).

7 Design alternatives

To address potential scalability problems with our centralized approach, a tree of controllers could be built that aggregates BARRIER_REACHED messages from children before sending a single message up the tree [18, 26, 37]. This tree could be built randomly from the expected set of hosts, or it could be crafted to match the underlying communication topology, in effect forming an overlay aggregation tree [34, 36]. In these cases, the master would send FIRE messages to its children, which would in turn propagate the message down the tree. One potentially difficult question with this approach is determining when interior nodes should pass summary BARRIER_REACH messages to their parent. Although a tree-based approach may provide better scalability since messages can be aggregated up the tree, the latency required to pass a message to all participants is likely to increase since the number of hops required to reach all participants is greater than in the centralized approach.

A gossip-based algorithm could also be employed to manage barriers in a fully decentralized manner [2]. In this case, each node acts as a barrier manager and swaps barrier status with a set of remote peers. Given sufficient pair-wise exchanges, some node will eventually observe enough hosts having reached the barrier and it will fire the barrier locally. Subsequent pair-wise exchanges will propagate the fact that the barrier was fired to the remainder of the nodes in the system, until eventually all active participants have been informed. Alternatively, the node that determines that a barrier should be released could also broadcast the FIRE message to all participants. Fully decentralized solutions like this have the benefit of being highly fault tolerant and scalable since the work is shared

equally among participants and there is no single point of failure. However, since information is propagated in a somewhat ad hoc fashion, it takes more time to propagate information to all participants, and the total amount of network traffic is greater. There is an increased risk of propagating stale information as well. In our experience, we have not yet observed significant reliability limitations with our centralized barrier implementation to warrant exploring a fully decentralized approach.

We expect that all single-controller algorithms will eventually run into scalability limitations based on a single node's ability to manage incoming and outgoing communication with many peers. However, based on our performance evaluation (see Section 6), the performance of centralized barriers is acceptable to at least 100 nodes. In fact, we find that our centralized barrier implementation out-performs an overlay tree with an out-degree of 10 for 100 total participants with regards to the time it takes a single message to propagate to all participants.

8 Related Work

Our work builds upon a number of synchronization concepts in distributed and parallel computing. For traditional parallel programming on tightly coupled multiprocessors, barriers [19] form natural synchronization points. Given the importance of fast primitives for coordinating bulk synchronous SIMD applications, most MPPs have hardware support for barriers [23, 31]. While the synchronization primitives and requirements for large-scale networked systems discussed vary from these traditional barriers, they clearly form one basis for our work. Barriers also form a natural consistency point for software distributed shared memory systems [6, 20], often signifying the point where data will be synchronized with remote hosts. Another popular programming model for loosely synchronized parallel machines is message passing. Popular message passing libraries such as PVM [15] and MPI [25] contain implementations of barriers as a fundamental synchronization service.

Our approach is similar to Gupta's work on Fuzzy Barriers [17] in support of SIMD programming on tightly coupled parallel processors. Relative to our approach, Gupta's approach specified an entry point for a barrier, followed by a subsequent set of instructions that could be executed before the barrier is released. Thus, a processor is free to be anywhere within a given region of the overall instruction stream before being forced to block. In this way, processors completing a phase of computation early could proceed to other computation that does not require strict synchronization before finally blocking. This notion of fuzzy barriers were required in SIMD programs because there could only be a single outstanding barrier at any time. We can capture identical semantics

using two barriers that communicate state with one another. The first barrier releases all entering nodes and signals state to a second barrier. The second barrier only begins to release nodes once a sufficient number (perhaps all) of nodes have entered the first barrier.

Our approach to synchronizing large-scale networked systems is related to the virtual synchrony [2, 3] and extended virtual synchrony [27] communication models. These communication systems closely tie together node inter-communication with group membership services. They ensure that a message multicast to a group is either delivered to all participants or to none. Furthermore, they preserve causal message ordering [22] between both individual messages and changes in group membership. This communication model is clearly beneficial to a significant class of distributed systems, including service replication. However, we have a more modest goal: to provide a convenient synchronization point to coordinate the behavior of more loosely coupled systems. It is important to stress that in our target scenarios this synchronization is delivered mostly as a matter of convenience rather than as a prerequisite for correctness. Any inconsistency resulting from our model is typically detected and corrected at the application, similar to soft-state optimizations in network protocol stacks that improve common-case performance but are not required for correctness. Other consistent group membership/view advancement protocols include Harp [24] and Cristian's group membership protocol [8].

Another effort related to ours is Golding's weakly consistent group membership protocol [16]. This protocol employs gossip messages to propagate group membership in a distributed system. Assuming the rate of change in membership is low enough, the system will quickly transition from one stable view to another. One key benefit of this approach is that it is entirely decentralized and hence does not require a central coordinator to manage the protocol. As discussed in Section 7, we plan to explore the use of a distributed barrier that employs gossip to manage barrier entry and release. However, our system evaluation indicates that our current architecture delivers sufficient levels of both performance and reliability for our target settings.

Our loose synchronization model is related in spirit to a variety of efforts into relaxed consistency models for updates in distributed systems, including Epsilon Serializability [30], the CAP principle [14], Bayou [32], TACT [38], and Delta Consistency [33].

Finally, we note that we are not the first to use the arrival rate at a barrier to perform scheduling and load balancing. In Implicit Coscheduling [11], the arrival rate at a barrier (and the associated communication) is one

consideration in making local scheduling decisions to approximate globally synchronized behavior in a multi-programmed parallel computing environment. Further, the way in which we use barriers to reallocate work is similar to methods used by work stealing schedulers like CILK [5]. The fundamental difference here is that idle processors in CILK make local decisions to seek out additional pieces of work, whereas all decisions to reallocate work in our barrier-based scheme are made centrally at the barrier manager.

9 Summary

Partial barriers represent a useful relaxation of the traditional barrier synchronization primitive. The ease with which we were able to integrate them into three different existing distributed applications augurs well for their general utility. Perhaps more significant, however, was the straightforward implementation of wide-area MapReduce as enabled by our expanded barrier semantics. We are hopeful that partial barriers can be used to bring to the wide area other sophisticated parallel algorithms initially developed for tightly coupled environments. In our work thus far, we find in many cases it may be as easy as directly replacing existing synchronization primitives with their relaxed partial barrier equivalents.

Dynamic knee detection in the completion time of tasks in heterogeneous environments is also likely to find wider application. Being able to detect multiple knees has added benefits since many applications exhibit multimodal distributions. Detecting multiple knees gives applications more control over reconfigurations and increases overall robustness, ensuring forward progress even in somewhat volatile execution environments.

References

- [1] D. G. Andersen, H. Balakrishnan, and F. Kaashoek. Improving Web Availability for Clients with MONET. In *NSDI*, 2005.
- [2] K. Birman. Replication and Fault-Tolerance in the ISIS System. In *SOSP*, 1985.
- [3] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *SOSP*, 1987.
- [4] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *CACM*, 36(12), 1993.
- [5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP*, 1995.
- [6] M. Z. Brian Bershad and W. Sawdon. The Midway Distributed Shared Memory System. In *CompCon*, 1993.
- [7] Computing TCP's Retransmission Timers (RFC). <http://www.faqs.org/rfcs/rfc2988.html>.
- [8] F. Cristian. Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems. *DC*, 4(4), 1991.
- [9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.
- [10] E. Dijkstra. The Structure of the "THE"-Multiprogramming System. *CACM*, 11(5), 1968.
- [11] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *SIGMETRICS*, 1996.
- [12] EMAN. <http://ncmi.bcm.tmc.edu/EMAN/>.
- [13] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. GGF, 2002.
- [14] A. Fox and E. Brewer. Harvest, Yield, and Scalable Tolerant Systems. In *HotOS*, 1999.
- [15] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *C-P&E*, 4(4):293–312, 1992.
- [16] R. Golding. A Weak-Consistency Architecture for Distributed Information Services. *CS*, 5(4):379–405, Fall 1992.
- [17] R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *ASPLOS*, 1989.
- [18] R. Gupta and C. R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *IJPP*, 18(3):161–180, 1990.
- [19] H. F. Jordan. A Special Purpose Architecture for Finite Element Analysis. In *ICPP*, 1978.
- [20] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX*, pages 115–131, 1994.
- [21] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.
- [22] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7), 1978.
- [23] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *JPDC*, 33(2):145–158, 1996.
- [24] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shriram, and M. Williams. Replication in the Harp file system. In *SOSP*, 1991.
- [25] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.
- [26] S. Moh, C. Yu, B. Lee, H. Y. Youn, D. Han, and D. Lee. Four-ary tree-based barrier synchronization for 2d meshes without non-member involvement. *TC*, 50(8):811–823, 2001.
- [27] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *CACM*, 39(4), 1996.
- [28] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *HotNets*, 2002.
- [29] Plush. <http://plush.ucsd.edu>.
- [30] C. Pu and A. Leff. Epsilon-Serializability. Technical Report CUCS-054-90, Columbia University, 1991.
- [31] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *ASPLOS*, 1996.
- [32] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*, 1995.
- [33] F. Torres-Rojas, M. Ahamad, and M. Raynal. Timed Consistency for Shared Distributed Objects. In *PODC*, 1999.
- [34] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *TCS*, 21(2), 2003.
- [35] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP*, 2001.
- [36] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *SIGCOMM*, 2004.
- [37] J.-S. Yang and C.-T. King. Designing tree-based barrier synchronization on 2d mesh networks. *TPDS*, 9(6):526–534, 1998.
- [38] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *OSDI*, 2000.

System- and application-level support for runtime hardware reconfiguration on SoC platforms

Dimitris Syrivelis and Spyros Lalis

*Department of Computer and Communications Engineering
University of Thessaly, Hellas*

Abstract

This paper discusses the design and implementation of a system-level mechanism and corresponding application-level support that enables programs running on a reconfigurable SoC to modify the underlying FPGA at runtime. Applications may request the addition and/or removal of softcore devices at any point in time. Requests are handled in a coordinated way via a separate user-level process that fetches the configuration bistream from an external server. System reconfiguration is implemented via a fast suspend-resume mechanism with support for dynamic softcore device address management to achieve flexible device placement on the reconfigurable fabric. Even though our approach does not rely on advanced (and expensive) hardware that supports dynamic partial reconfiguration, the obtained functionality is sufficient for a wide range of application scenarios.

1 Introduction

The technology of field-programmable gate arrays (FPGAs) has the potential to change the way computing systems are being built. While FPGAs are not as fast or energy saving as corresponding ASICs [11] they have the considerable advantage of flexibility: it becomes possible to reconfigure a system not only in terms of software but also in terms of underlying hardware support. In order to exploit this potential one faces challenging issues, such as codesigning hardware and software components, and seamlessly deploying hardware logic on platforms.

In this context it is of particular importance to support a flexible yet robust runtime reconfiguration, allowing for the dynamic downloading and installation of new softcore components. This opens the way for a wide range of possible application scenarios regarding automated system upgrades and customized platform (re)configuration. For example, one may introduce several hardware/software codesigned components that em-

ploy customized hardware codecs and accelerators to offload the CPU, boost performance and lower power consumption. The system could also decide which modules fit concurrently on the reconfigurable fabric and select the most appropriate combination, based on the current state and explicitly provided specifications. Even more radical adaptation can be realized on systems with a softcore CPU, in which case it becomes possible to add mechanisms that track CPU usage and create application execution profiles. This information can in turn be exploited to fine-tune specific CPU components as well as to select the most beneficial combination of application-level hardware accelerators. Notably, the efficient online profiling for softcore CPU platforms investigated in [14] could provide the basis for such work.

Runtime reconfiguration in essence translates to *transparency*, i.e. the ability to maintain system and application state so that execution may proceed after (or even during) system reconfiguration without the need for a restart/recovery procedure. Compared to platforms where the FPGA is merely a peripheral of the CPU, this is harder to achieve in a system-on-chip (SoC) because the entire system and application runtime state resides within the reprogrammable fabric itself. Specifically, in order for the runtime state to be kept intact, the FPGA hardware must: (i) support partial reconfiguration; (ii) retain the main softcore logic active while it is being reconfigured; (iii) offer the means for self-controlling the reconfiguration process [22]. For the time being, FPGA vendors provide these features only in expensive product families, and even these devices have constraints in terms of the dynamic partial reconfiguration (DPR) that can be achieved in practice. For this reason approaches that rely on advanced FPGA hardware are not suitable for cheap commodity platforms, or systems with considerable reconfiguration requirements that cannot be implemented given the current limitations of DPR.

In this paper we present work on achieving runtime reconfiguration for SoC platforms featuring a softcore

CPU, without relying on advanced FPGA features. Our goal is to let applications add and remove softcore devices dynamically. The main contributions of this paper are: (1) the introduction of a system-level mechanism and application-level support for reconfiguring a SoC platform at runtime, (2) an implementation that runs on an off-the-shelf embedded device, and (3) a proof-of-concept demo system. We underline that our approach is entirely implemented in software, thus does not achieve the same functionality that is (theoretically) possible via DPR. It nevertheless provides considerable runtime flexibility that is sufficient for most conventional application systems.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 gives an overview of our approach. Section 4 describes a concrete system-level implementation based on the uClinux Xilinx Microblaze port. The corresponding application-level support is described in Section 5. Section 6 presents an extension of our scheme to achieve an integrated handling of softcore devices and physical off-chip peripherals. Section 7 discusses performance related issues. Section 8 presents a proof-of-concept demo setup. Finally, Section 9 concludes the paper.

2 Related Work

Significant efforts have been done on hardware-software codesign to exploit the potential of reconfigurable hardware. Researchers try to build a unified and transparent programming model as well as a standard interface for the integration of hardware accelerators independently of the underlying platform details. A methodology for codesigning applications along with corresponding development tool support is presented in [16]. It proposes a binary level hw/sw partitioner that takes as input a software binary, decompiles it to recover high-level information, determines the regions that should be implemented in hardware (using appropriate profiling information) and generates modified binaries that have the critical code fragments replaced by instructions that access the hardware versions. In [18] a high-level programming model is proposed based on a virtualization layer through which softcore devices can be accessed in a transparent way. Both approaches assume that the underlying platform provides appropriate dynamic reconfiguration support, allowing for arbitrary modules to be added at runtime. This is far from straightforward to achieve in reconfigurable SoC platforms where the CPU itself occupies an area of the reconfigurable fabric.

A lot of research has been conducted on FPGA architectures and development tools for dynamic reconfiguration support. The first step has been to enable partial reconfiguration through corresponding partial bitstream

generation tool capabilities [5][9][13], then to change FPGA architecture design so that it can retain the rest of the logic active while it is being partially reconfigured [2][3][15][22]. It must be stressed that dynamic partial reconfiguration (DPR) is still an active field of research. For the time being there are several problems [22] which make it hard or even practically impossible to apply DPR, especially for large and complicated designs such as SoC platforms that feature a softcore CPU: the partially reconfigurable FPGA area placement and size; the external IOB routing constraints that enforce the whole FPGA board layout to be designed with DPR scenarios in mind; and –last but not least– the limited number of Tristate Buffers (TBUFs) that must be used to interconnect dynamically loaded modules with the rest of the logic [22].

Considerable work has been done to support the runtime reconfiguration on SoCs or platforms featuring a separate CPU. This typically concerns mission-specific platforms, or is integrated within a proper (embedded) operating system context. We briefly discuss indicative systems representing a variety of different approaches.

A typical framework for achieving dynamic reconfiguration of a dedicated SoC based on DPR is presented in [7]. Part of the FPGA is used for a fixed softcore control subsystem which communicates with a remote host. The rest of the FPGA is used to place custom logic. Reconfiguration can be triggered by the remote host, at any point in time, which sends the corresponding bistream to the control unit. The bitstream can also be encrypted for security purposes. This approach is suitable for single-application systems.

In [20] runtime DPR support is provided for a SoC featuring a softcore CPU and an embedded operating system. A dedicated kernel-level driver is introduced which provides raw access to FPGA configuration data, allowing it to be modified in an online fashion. This interface can be used by applications or shell scripts to change part of the FPGA at runtime. However, only simple reconfiguration scenarios can be implemented given the limitations of DPR.

A different approach is employed in [12] where the FPGA is pre-partitioned into a fixed number of custom softcore units, and an extra layer is used to provide the abstraction of unit allocation. The program loader distinguishes between software and softcore tasks and dynamically links the former with a free softcore unit. This approach has been implemented in a system with a separate (hardware) CPU. It can be used to eliminate some DPR constraints for a specific platform, but reduces flexibility. This is because it partitions the FPGA to an a priori defined number of nodes that communicate with each other via a fixed interconnection architecture. It is thus impossible to dynamically install arbitrary hardware compo-

nents that are customized for different applications.

Task-based reconfiguration using a suspend-resume model for a multi-node architecture is presented in [8]. When a node needs to reconfigure, its tasks are suspended and restarted on another node. During this migration, hardware functions may be mapped to software versions thereof, depending on the resources available on the destination node. When reconfiguration completes the original node can be re-assigned its old tasks and proceed with their execution. This approach enables a full reconfiguration of a SoC node, but requires at least two nodes. It has also been implemented using customized hardware and a separate softcore CPU.

Our reconfiguration scheme is geared towards SoC platforms with a softcore CPU and an embedded operating system but does not rely on DPR (merely an off-chip reconfiguration circuit is required). It constitutes a practical option for achieving runtime reconfiguration on top of cheap FPGA systems, without DPR functionality nor any special support from the softcore development toolchain. We employ a suspend-resume model for the entire SoC, but a node can autonomously perform the entire reconfiguration without requiring a second node that must act as its slave. The proposed approach maintains application and operating system state during reconfiguration and lets drivers initialize or even re-establish the state of softcore devices after reconfiguration completes.

Given that we target primarily resource constrained platforms, the hardware configuration bitstreams have to be retrieved from a remote server over the Internet. This is similar to the Xilinx Online (Internet Reconfigurable Logic) framework [21], which introduces a remote hardware-update capable facility on top of an operating system. The difference is that in our case it is the user applications that trigger a reconfiguration, not the remote server. It is in fact possible for any process to request a platform reconfiguration at any point in time yet in a controlled way that ensures graceful degradation in case of resource shortage. Moreover, our approach transparently maintains system and application state across reconfigurations.

3 Approach overview

The goal of our work is to support runtime reconfiguration for SoC platforms that feature a softcore CPU. Specifically, we wish to let applications dynamically add and remove softcore devices that can be accessed via a fast bus or memory mapped I/O. For example, special hardware accelerators, bus drivers and controllers for external hardware, or extra CPU softcore units, could be installed on demand, according to the requirements of the applications running on the system. Again, we stress that this functionality is to be achieved without relying

on DPR capable hardware and corresponding partial bitstream generation tools support. The next subsections give an overview of our approach, motivating the various decisions taken.

3.1 The Concept

Our approach is based on a suspend-resume technique, as follows. In a first step, before the actual reconfiguration process begins, the FPGA bitstream corresponding to the new hardware layout for the entire SoC is stored in external memory (we do not address the computation of the bitstream per se). Then, the system saves its current runtime state and initiates FPGA programming. When this completes, the system restarts and control goes to the first stage loader. This checks whether a reconfiguration took place, in which case it overrides the default boot sequence, restores the saved system state and adjusts basic system device information. Finally, prior to resuming normal execution, the device drivers are notified in order to handle the side-effects of FPGA reconfiguration; most notably to initialize / restore the state of the devices. A schematic illustration of this process is given in figure 1.

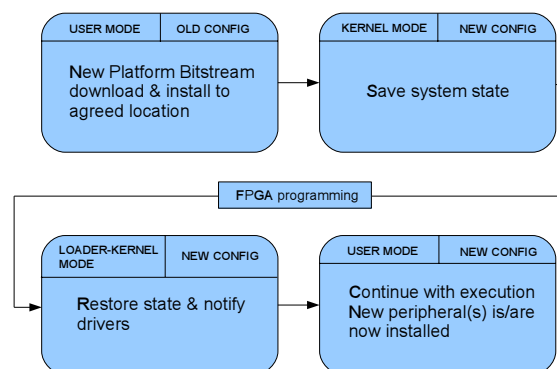


Figure 1: The main phases of the reconfiguration scheme

Despite the fact that the entire FPGA is programmed from scratch in a conventional fashion, the reconfiguration flexibility provided to the application level is comparable to what would have been possible using techniques that rely on DPR. We note that, in principle, the same scheme could also be used to enable a radical modification of the softcore CPU itself (changing the softcore CPU characteristics according to application workload has been shown to boost performance [6]). However, our approach cannot be applied if part of the FPGA logic is required to remain active during reconfiguration, e.g. for hard real-time applications.

3.2 Device Address Assignment

Given that peripheral devices can be added and removed dynamically, the management of device addresses (more specifically, channel ids for devices that are accessed via a fast bus or specific addresses in the case of memory mapped I/O) becomes a central design issue.

The “obvious” approach of a priori assigning each softcore device a fixed address is not attractive. In the case of fast bus access this would considerably limit the number of devices that can be supported because only a few different channel ids are typically supported in such architectures. This holds to a far lesser extent for memory mapped I/O, but then again the corresponding address range (though large) is not infinite. Thus an artificial upper bound for the number of peripheral devices that can be considered is introduced in this case too. What’s probably worse, to avoid conflicts, some central authority or service would be required to assign channel ids and address ranges to each softcore device being (ever) developed.

It is possible to eliminate these drawbacks by assigning addresses dynamically, when a device is first installed in the system. Still, in this case, each time the system reconfigures, the new platform memory layout would have to be computed based on the current configuration and so as to ensure that the addresses of all devices that continue to be a part of the new configuration remain valid. This implies that the new system image must be produced in an online fashion, taking such constraints as input.

To maximize flexibility, we do not require device ids and addresses to remain fixed across system reconfiguration(s). This decouples the process of computing the new system image from any dynamic constraints, other than the type and number of softcore devices that need to be placed on the FPGA. Furthermore, rather than having to compute bitstreams on demand, it becomes possible to exploit databases of pre-fabricated perhaps even highly optimized hardware layouts that could be maintained by device manufacturers.

3.3 Device Access Transparency

Since device addresses are not a priori known and may change in the midst of program execution, additional support is required so that applications are able to access softcore devices in a reliable fashion.

The desired access transparency and safety at the application level could be achieved via a device address translation and checking mechanism, in the spirit of a virtual memory management unit. This would have required a non-trivial modification of the softcore architecture, which is beyond the scope of our current work.

For this reason we adopt a more restricted but yet

comparably functional solution, by requiring applications to access peripheral devices via corresponding drivers. Device drivers are a natural way for introducing new hardware functionality in a structured fashion. This also guarantees that applications access softcore devices in an explicit fashion and under system control. Last but not least, device access transparency can still be achieved provided that drivers offer suitable *reconfiguration-transparent* primitives to the application.

4 Implementation of system-level support

This section presents the implementation details of our reconfiguration scheme, for the case of a concrete embedded device, softcore architecture and runtime system. We also discuss issues concerning the development of device drivers in order to deal with the dynamics of reconfiguration.

4.1 Platform

System-level support for our reconfiguration scheme has been integrated into the Microblaze-uClinux kernel port [19] that runs on top of the corresponding MMU-less softcore architecture. Microblaze utilizes Harvard-style separate instruction and data busses which conform to IBM’s CoreConnect On-Chip Peripheral Bus standard. Bus arbiters can be automatically instantiated, permitting the instruction and data busses to be tied together in order to create conventional von Neumann-style system architectures.

The host Platform is an Atmark Techno Suzaku [1] (Figure 2) featuring a Xilinx Spartan-3 (XS3C1000) FPGA along with off-chip 16MB SDRAM, 8MB flash, a MAC/PHY core and a configuration controller. The main on-chip softcore modules are the Microblaze core with local memory, Onboard Peripheral Bus, Local Memory Bus, Fast Simplex Links Bus, system timer, interrupt controller, SDRAM controller and an external memory controller.

FPGA configuration is initiated and controlled via Select Map by the external controller and the bitstream is stored in an external flash memory. The reconfiguration procedure can be initiated both by hardware (during power up) and software (write 0x1 to a special register) means. Notably, the power supply is not cut-off during reconfiguration and that the SDRAM data are *not* corrupted because the chip supports self-refresh.

4.2 The Peripheral Device Location Table

As discussed in the previous section, devices may change their addresses after each reconfiguration (with the ex-

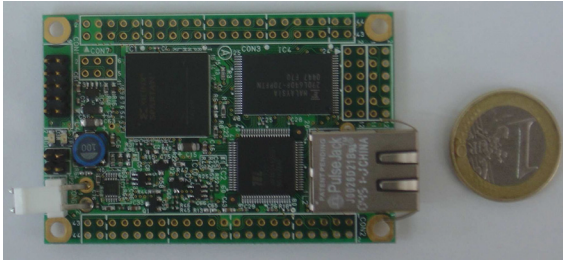


Figure 2: Atmark Techno Suzaku

ception of the execution and data memory controller which are mapped at a specific location because code and data are statically linked to fixed addresses). This means that drivers must be given a mechanism for retrieving the device addresses that are valid at any point in time.

For this purpose the kernel is augmented with the so-called Peripheral Device Location Table (PDLT), an array that contains the addresses of the devices that are available in the current configuration. Each device is assigned a globally agreed offset in the PDLT that is known to program developers. For convenience, we define these offsets based on the well-known major and minor numbers combination of device drivers in the Linux kernel. A PDLT of a few Kilobytes is sufficient to accommodate a large number (thousands) of different devices; of course, the number of devices that can actually co-exist in a system configuration (FPGA image) is limited.

Drivers must be programmed to retrieve the current device base addresses from the corresponding PDLT locations. A zero value implies that the device is not available in the current configuration. The PDLT contents are also exported in user space through the */proc* filesystem so that applications can check the current platform configuration for a particular device.

When a reconfiguration takes place, the contents of the PDLT are updated by the first stage loader during system resume. The loader is programmed into softcore processor local memory and is stored in the configuration bitstream, together with the PDLT entries of the current system layout. Updating the PDLT in kernel space therefore requires a simple copy operation. Since the location of the PDLT depends on the kernel configuration, its start address is stored into a well defined non-volatile memory location so the bootloader can access it.

The bootloader is build using the Board Support Package tool of the Xilinx EDK (Ver 6.3) environment, which generates C `#define` preprocessor directives with BaseAddresses, thereby making the process of generating the PDLT and storing its contents in the configuration bitstream quite simple. It would also be possible to enrich the Xilinx development environment with scripts

that automate this task; though we have not done this.

4.3 Triggering Reconfiguration

Reconfiguration is triggered via a special system call that executes as follows. First, interrupts are masked and the old interrupt mask is stored in a local variable. Then all pending interrupt bottom halves are executed by waking up the linux kernel *ksoftirqd* daemon. The timer bottom half is excluded from this process because it may result in a context switch. Susequently the Interrupt Vector Table and relevant machine registers are stored in a designated area in the kernel image in DRAM (instead of saving the current value of the Instruction Pointer, the address of the resume function is stored). At this point the external controller is triggered to initiate FPGA programming. When this completes, control goes to the bootloader which retrieves the state saved via the system call, calculates the PDLT address in kernel memory, copies its contents from the image, and restores the Interrupt Vector Table and registers. Finally, control returns to the system call context, and the device drivers are notified (see next section) before restoring the interrupt mask and proceeding with the execution of the process that invoked the call. The entire procedure takes less than a second to complete on our platform.

A drawback of letting the system state reside on DRAM is that after a power reset the system reverts to its “original” state and configuration. For this reason, we have implemented a so-called hibernation option. In this case, the system image is copied from DRAM to non-volatile storage (flash) before initiating FPGA programming. The reconfiguration mode (normal vs hibernation) is specified as a parameter of the system call and is stored along with the rest of the runtime state. This information is retrieved upon restart by the bootloader, and if reconfiguration was performed in hibernation mode, the system image is restored into DRAM prior to continuing with the default resume action sequence.

While the hibernation option enhances robustness, it also introduces a significant delay. The total time needed to dump the DRAM image on flash is well above 30 seconds for our platform. Our backup scheme is simple (e.g. the flash is written in polling read mode since all interrupts are disabled) and lacks advanced features, such as checkpointing. Faster non-volatile media and more elaborate I/O operations could reduce this delay, but this could also lead to inconsistencies with respect to the state being saved, in which case more sophisticated hibernation mechanisms [4] may have to be employed.

Notably, our approach is not directly applicable on systems that are interfaced to complex hardware. In this case, a system-wide quiescing of user space processes and kernel thread activity would be required, both prior

and after the system suspend sequence so that the state of peripherals can be properly saved and restored, respectively.

4.4 Device Driver Notification

When the system reconfigures, all devices are destroyed, and then re-installed, possibly in a different area within the FPGA fabric; and in a state that most likely requires further initialization before the device becomes operational. As a consequence, even though the device addresses are properly stored in the PDLT, additional device driver specific repair actions may be needed in order to preserve the continuity of device usage at the application level.

For this purpose device drivers may register a so-called *reconfiguration handler*, which is invoked by the kernel after reconfiguration, before returning control to the application. This routine can be used to perform various housekeeping tasks, such as to initialize the device to a default operating mode, perhaps even restoring it to a previous state, and abort pending operations whose execution may have been compromised due to the FPGA reconfiguration. Device drivers that do not require any initialization/restoration actions need not provide a handler. A simple priority scheme is used to enable the execution of handlers in a certain order.

In our current system prototype we have successfully implemented reconfiguration handlers for the UART, Ethernet, flash, GPIO, interrupt controller and system timer drivers. Since our platform has a softcore timer, each reconfiguration introduces a real-time clock lag (noticeable from an external observer). This error could be corrected by measuring the (fixed) amount of time required for the system to reconfigure, and letting the timer driver increment the system time by this value after each reconfiguration.

4.5 Reconfiguration-Transparent Drivers

Application programs should access devices without caring about reconfigurations that may take place during their execution. Put in other words, device drivers should offer *reconfiguration-transparent* operations. Although the specifics of how to achieve satisfactory functionality are highly device-dependent, we have found the following guidelines to be of use for most cases.

Upon startup the device driver initializes its internal state as well as the device, as usual. When the reconfiguration handler is called, the device is initialized so that it can be properly accessed via the driver operations. Moreover, all processes that have been suspended inside a driver operation are resumed. This implies that the reconfiguration handler must be able to access all driver

specific wait queues used by blocking operations, which can be typically achieved via a global wait queue list.

Each driver operation retrieves the device address from the PDLT and uses it to access the device. Notably, the PDLT entry may contain a zero value, indicating that the device is not installed in the current configuration, in which case the driver operation returns an error (e.g. ENODEV). Moreover, configuration parameters and/or additional internal state are recorded so that the device can be properly initialized via the reconfiguration handler. Blocking operations also maintain global state that is used to determine, when they eventually resume, whether a reconfiguration took place in the meantime. If this is not the case, the operation proceeds as usual. Else, the device address is retrieved from the PDLT and the operation can be re-tried.

For the sake of completeness we note that some devices may be *asynchronous*, in which case the effects of driver operations do not necessarily take place within the context of the respective invocations. Moreover, it may be impractical or even impossible for the driver to maintain the device's internal state so that it can be restored. In this case, reconfiguration could lead to state loss, violating transparency. This could be avoided by introducing a locking scheme that allows a driver to block (a requested) reconfiguration until all such operations are acknowledged by the softcore device. We plan to address this issue in a future version of our implementation.

5 Application-level support

The described system-level support enables applications to trigger reconfiguration at any point in time according to their needs. However, it is undesirable to give applications such direct control over the system's resources. One problem is that some applications use devices merely as a performance enhancement option, whereas others may be unable to operate without the requested devices being available. If there are not enough hardware resources to accommodate all devices, precedence should be given to the ones that are vital to application execution. This also implies the removal of devices that are currently installed but not of vital importance to the applications using them. Another issue is that concurrently running applications will trigger multiple consecutive reconfigurations, even though in some cases the same result could be achieved more efficiently, via a single reconfiguration.

This functionality cannot be achieved if each application is allowed to reconfigure the system while being oblivious to the needs of others. With this motivation we do not allow the reconfiguration call to be invoked from within user processes, and instead introduce a separate mechanism through which reconfiguration is triggered in

a coordinated way that ensures maximum overall performance.

5.1 API and background processing

To control reconfiguration according to system-wide policies, applications send device addition and removal requests to a system process with root privileges, called the reconfiguration daemon. The corresponding API (shown below) is implemented as a shared library and communicates with the daemon via unix domain sockets.

```
#define DEV_RMV 0
#define DEV_ADD 1
#define DEV_MND 2

struct dev_req {
    char dev_name[64];
    int actionflags;
};

int device_request(struct *dev_req, \
                  int nofreqs);
```

Applications may use the *device_request* call to issue one or more device addition and/or removal requests. Each request contains the device name (the file name of the corresponding kernel driver) and the action to be performed (the DEV_ADD and DEV_RMV flag is used to specify device addition and removal, respectively). An addition can be specified as mandatory (via the DEV_MND flag) indicating that the device is needed for the application to perform properly.

Requests are processed asynchronously and notification occurs via a SIGRECONF signal. This signal is sent to processes that issued an optional addition request which was satisfied. It is also sent after *each* reconfiguration to processes that requested a mandatory addition, even if this was not satisfied; allowing them to take corrective action or abort. Applications may catch the SIGRECONF signal in a conventional manner, by registering a handler which can determine the presence of the requested device via the */proc* file system.

The reconfiguration daemon maintains a list of requests issued by applications, each carrying the id of the sender process and status information (pending or applied). When a new request arrives, the daemon inserts it in the list and waits for more requests to arrive. If no new request arrives within a given time threshold, the list contents are combined to produce the new platform configuration. In case it is not possible (due to resource constraints) to satisfy all addition requests, these are considered in a first-come-first-serve order, and priority is

given to the mandatory requests. When a feasible configuration has been determined, the daemon writes the corresponding FPGA bitstream to the designated memory area and triggers reconfiguration via the system call. It then updates the status of the requests to reflect the current configuration and notifies application processes via a SIGRECONF signal. As a trivial optimization, removal requests do not lead to a reconfiguration unless the list contains at least one pending (and feasible) device addition request.

A process that has issued an addition request may terminate without issuing a corresponding removal request. For this reason the daemon periodically checks (through the */proc* filesystem) the liveness of all processes that issued addition requests. If a process terminates and its addition request has not yet been applied, it is removed, else a corresponding removal request is added to the list to ensure that the device that has been added by this process will be removed. Moreover, at this point it is also convenient for the daemon to remove the kernel drivers, that were used by processes that are no longer alive, since they will no longer be useful in the new configuration.

5.2 Example

This functionality is illustrated in Figure 3 for an indicative scenario. The application processes, softcore devices and request list maintained by the reconfiguration daemon are shown for each step. The eclipses on the top left area denote running processes that issue reconfiguration requests. The installed softcore devices are represented by rectangles in the top right area. The request list is depicted in the bottom part, showing for each entry its process id (upper left), device name (lower left), action type (lower right) and status (upper right, where white and black color stands for pending and applied, respectively). For simplicity, we assume that all addition requests are flagged optional.

We briefly discuss each illustrated step in the following. Initially (a) there are three processes, P1, P2 and P3. At some point in time P1 issues a request to the reconfiguration daemon for the addition of device D1, and P3 issues a request for device D3. Assuming that both devices can be accommodated using the available hardware resources, these are installed via a single reconfiguration (b). Later on (c) P1 terminates (a remove request for D1 is added on its behalf) and P2 issues a request for device D2, leading to a new configuration where D2 is added and D1 is removed (d). Then P4 requests the addition of device D4 (e), but assuming there are not enough hardware resources no reconfiguration takes place. Eventually (f) P2 requests the removal of D2, making it possible to install a new configuration with D4 (g). Finally (h) P4 terminates and, as a result, a remove request is added on

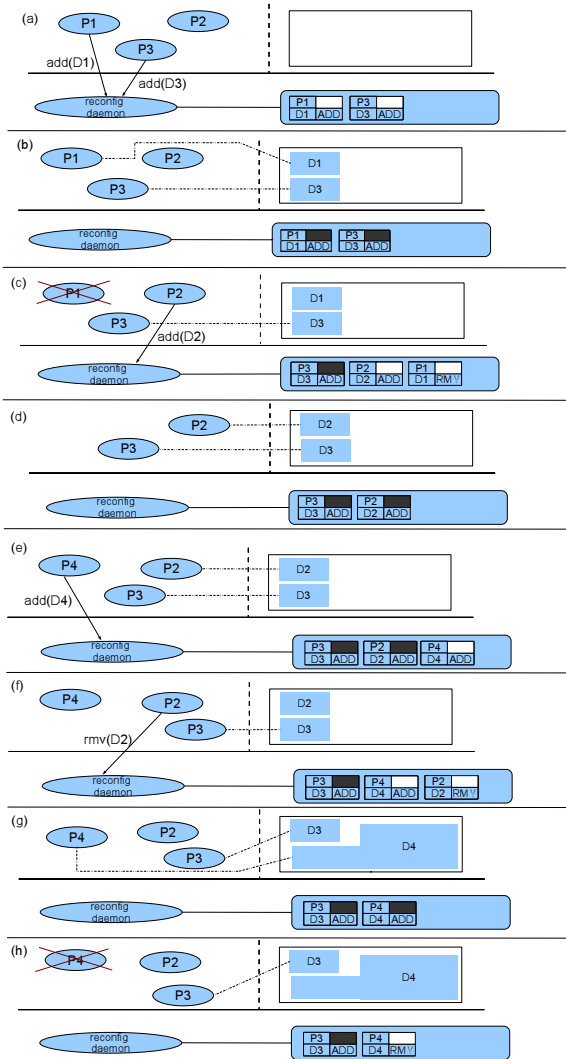


Figure 3: A reconfiguration scenario

its behalf, but no reconfiguration occurs (yet) since there are no pending addition requests to be satisfied.

5.3 Application-level transparency

Applications that rely on basic platform features (e.g. CPU, RAM, Ethernet) run safely on our system. They can be executed without any modification, and remain unaffected despite the (repeated) system reconfigurations that may take place at runtime.

If however a program wishes to use a custom software device, it must be implemented accordingly. To begin with, it must explicitly issue a device addition request and register a reconfiguration handler that will result in the desired adaptive behavior, e.g. start exploiting the device as soon as it becomes available. Once a de-

vice is added to the current configuration, transparency is achieved if (a) the device is mandatory and (b) it is accessed via a reconfiguration-transparent device driver. Else, a program may fail to access the device due to its relocation or removal from the FPGA; and should be prepared to deal with this case in a robust way.

We note that the addition of a device must be explicitly requested, even if it already exists in the current platform configuration. This is to let the reconfiguration daemon keep a correct reference count for each device. Also, given that device references are kept in user space, these are not inherited from parent to child processes and kernel-level threads. Thus separate device addition requests must be issued on behalf of each execution context, independently of whether this has already been done by the parent.

5.4 Remote bitstream fetch

To achieve a clean separation of concerns the FPGA bitstream of the desired platform configuration is provided by a separate process, called the bitstream server. The communication between the reconfiguration daemon and the bitstream server is over TCP/IP, hence the server can be conveniently placed on a remote host; which is particularly useful in the case of resource constrained platforms.

When the daemon wishes to reconfigure the system, it sends to the bitstream server the list of optional and mandatory devices that may need to remain or become available, respectively. Based on this input, the server replies with the configuration that can be implemented given the amount of hardware resources available (perhaps depending on other limitations as well) and a corresponding bitstream url. The daemon then downloads the bitstream from the server using the netflash utility [17].

The underlying working assumption is that the bitstream server knows the host platform details and has access to a database of pre-fabricated configuration bitstreams. For example, it could be a platform vendor service responsible for providing fully tested and highly optimized configurations. In principle, it would also be possible to integrate the bitstream server functionality with the hardware development toolchain so as to be able to synthesize new platform configurations on demand. Given that this task is quite time consuming (10 minutes approx. on a PC), this is not a very attractive solution for the time being.

6 Support for off-chip peripherals

Functional units requested by applications may require not only a software module but also additional off-chip peripherals, e.g. a sensor. In this particular case it makes

no sense to add the softcore module unless the peripheral is also physically connected to the system. Wishing to unify the softcore and physical aspect of peripherals, we extended the reconfiguration mechanism to handle application requests and the asynchronous event of a peripheral plug-in in an integrated fashion.

6.1 The hotplug detector

To accomplish this we introduce a special softcore device, the so-called hotplug detector. Its role is to capture the fact that external hardware has been connected to or disconnected from the system, respectively. In our prototype we allow up to 4 peripherals to be simultaneously plugged on the Suzaku board.

The corresponding module amounts to 1% of our FPGA resources. It is hooked on the Microblaze On-Chip Peripheral bus and is accessed through 4 memory mapped registers. Eight of the least significant bits of each register are connected to external I/O FPGA pins while the rest are grounded. We assume that an off-chip peripheral will be attached to the pins of a register, and will redirect Vcc and Gnd to form a code that uniquely identifies it (in our implementation we require this to be the major number of the corresponding kernel driver). We also expect Vcc to be redirected to the peripheral interrupt line which is connected to an external I/O pin as well. An illustrative schematic is shown in Figure 4.

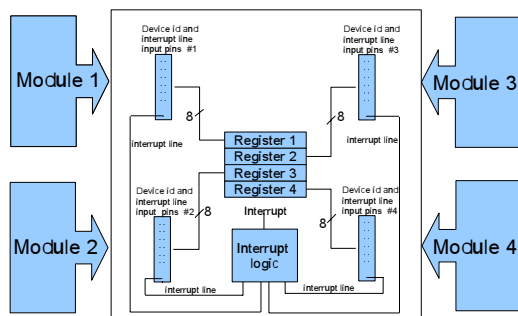


Figure 4: The hotplug detector high-level schematic

Access to the hotplug detector is provided via a character device driver, which supports the standard file operations interface as well as the *select* and *poll* system calls. The driver also registers an interrupt handler that is invoked when an off-chip peripheral is connected to and disconnected from the interface pins. The read operation is blocking and waits for an interrupt to occur.

The memory mapped registers have the value of zero when no peripheral is hooked and the driver remembers previous register states so it can determine whether a pe-

ripheral is connected or disconnected. When an interrupt occurs the Interrupt Service Routine scans all registers to determine which one has changed value, reads its contents and unblocks any waiting processes. Subsequent read operations then return the device id, the register number of the pin region, and a value (zero or one) indicating whether the device is connected to or disconnected from the system. To discover peripherals that have been hooked on the platform before starting the reconfiguration daemon (or powering up the system), the hotplug registers are examined via the *ioctl* system call when the daemon starts.

In our implementation we tried to avoid a complex hardware design that consumes a significant amount of resources. This is because we want to keep the hotplug detector as a basic platform feature that will be included in every configuration. By keeping state information in the device driver, rather than the hardware logic, we are also able to achieve reconfiguration transparency for the hotplug driver. Admittedly, using a 8 pin interface for device identification is a waste of external I/O resources. In principle one may use just 1 pin but this requires a more sophisticated communication protocol; see[10] for a similar DPR-based implementation.

6.2 Unified reconfiguration handling

The hotplug detector is accessed by the reconfiguration daemon to receive information about the addition and respectively removal of an off-chip peripheral. This information is then sent to the bitstream server, along with the contents of the request list.

It is the responsibility of the bitstream server to determine the possible layouts that may be installed on the FPGA, also taking into account the dependencies between softcore devices and off-chip peripherals. More specifically, a pending addition request for a softcore device that requires an off-chip peripheral is considered only if the peripheral is connected to the system. This decision naturally belongs to the bitstream server (rather than the reconfiguration daemon) since in our model it is the former that has access to platform-specific implementation data.

Once a mandatory softcore device that relies on a peripheral is installed, it is not automatically removed even if the required peripheral is disconnected from the system. In this case, the application will simply receive an error from the corresponding device driver. It may then explicitly request the removal of the device or terminate. On the other hand, the application may wish to keep the softcore device installed, expecting the peripheral to be re-connected to the system; this may involve out-of-the-loop information (e.g. the user's intention) which is not available to the low-level system mechanisms such as the

reconfiguration handler.

6.3 Example

Figure 5 gives a scenario illustrating this additional functionality (employing the same visual metaphors as in the previous example). Note that the hotplug detector module is considered to be already installed as a mandatory device.

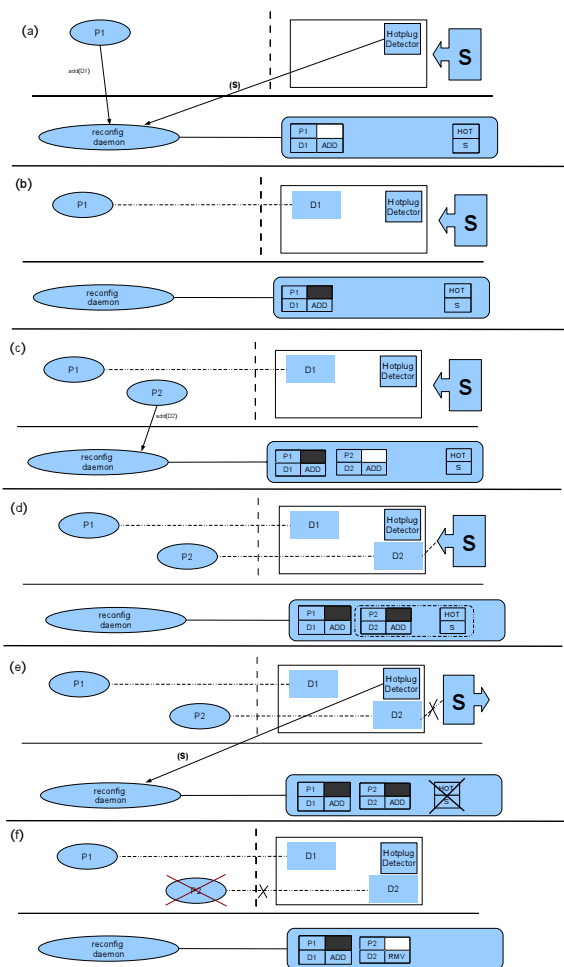


Figure 5: A reconfiguration scenario with hotplug event

Initially (a) process P1 requests the addition of device D1. At the same time, the user plugs in sensor S (that can be used only via device D2). The hotplug detector informs the reconfiguration daemon, which in turn adds a corresponding presence entry. The system then reconfigures and D1 is installed (b). After a while (c) a new process P2 starts, and requests the addition of device D2 (which requires the S). Given that S is already connected to the system, the system will reconfigure and D2 will be installed (d). Later on (e) S is disconnected from the

system, leading to a corresponding update of the reconfiguration daemon, but D2 remains installed. Finally (f) P2 receives an error from D2 (which tries to access S without success) and terminates (a removal request for D2 is added on its behalf).

7 Performance considerations

Since applications exploit softcore devices via kernel device drivers, each access operation comes at the cost of a system call. Each driver operation must also retrieve the base address of the device via the PDLT. This amounts to one extra instruction compared to the code that would have been generated using a fixed address scheme. A second instruction is needed for each different base-relative address used within a driver operation. We believe that this overhead is reasonable given that our approach is implemented in software, without requiring a modification of the softcore CPU architecture.

In our current implementation platform and setup, switching to a new configuration takes about 12 seconds to complete from the moment a process issues a device addition request (assuming the reconfiguration daemon does not wait for other requests to arrive). This delay is decomposed as follows. The communication with the bistream server including the download of the FPGA bitstream takes about 1.5 seconds (over a 10 Mbit Ethernet). Writing the bitstream on flash takes about 9 seconds. Finally, performing the reconfiguration system call (saving state, programming the FPGA, restarting and notifying drivers) takes less than 1 second. It is important to note that application processes continue their (concurrent) execution during this amount of time, except for the last step, i.e. the execution of the reconfiguration system call.

These figures are obviously specific to our implementation platform. The FPGA programming delay, for example, could grow for larger platforms; though these also tend to support higher programming speeds. What is more important, if it were possible to program the FPGA directly from DRAM (rather than requiring the bitstream to be copied on flash), the total reconfiguration delay (including the bitstream download from the network) could shrink to about 2-3 seconds.

8 Demonstration

To demonstrate our implementation we have developed a simple environment that comprises two different applications: a mandelbrot calculation and an audio signal monitor. Both applications are structured in the form of a client-server pair. The servers run on the Suzaku board as conventional application processes. The clients run on a PC providing a graphical user interface for controlling

the servers. Client-server communication is over TCP/IP and a LAN to which the Suzaku is connected via its Ethernet adapter. A schematic of the various components is given in Figure 6. A picture of the Suzaku board setup is shown in Figure 7.

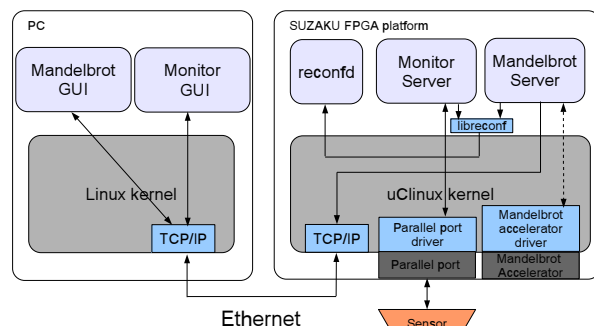


Figure 6: Demo Setup

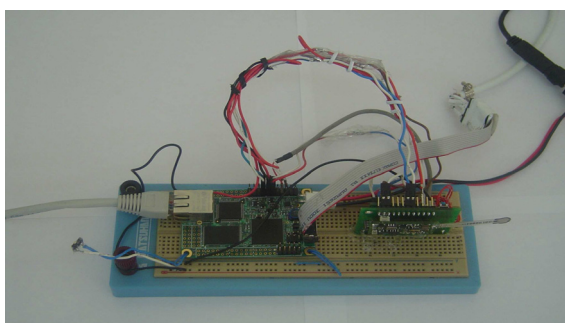


Figure 7: Suzaku board setup

8.1 The mandelbrot application

The mandelbrot client is used to send the computation parameters to the server and display the results produced. Moreover, it can be used to request the addition or removal of an accelerator module that is exploited by the server to perform the computation faster.

The server waits for incoming requests, performs the computation and sends the results back to the client. It is initially in a default state, performing the computation without relying on the accelerator module. When it receives a client command to add the accelerator, it issues a corresponding request, and enters an optimized mode of computation as soon as the softcore module is added to the system. Similarly, the server issues a removal request when it receives a corresponding client command. It continues however to opportunistically exploit the accelerator until the driver returns an error; indicating that the module has been actually removed.

As expected, hardware acceleration boosts performance both in terms of time and power consumption. Notably, our softcore CPU does not have a floating-point unit, hence the software version of mandelbrot uses the integer-based floating point operations of the gcc library. Figure 8 depicts the average energy needed to perform a certain computation for the software-only versus the hardware accelerated version. The average power consumption of the system in idle mode is used as a reference. Specifically, the hardware-based version requires about 7,6% of the energy the software version needs and is 7,33 times faster (labels 2 and 4 vs 1 and 5). The former version includes the extra delay and power consumption for downloading the bistream and reconfiguring the system *before* initiating the computation (labels 2 and 3). The hardware-based version requires about 5,6% of the power that the software version consumed and is 10 times faster in case the accelerator is already installed (labels 3 and 4).

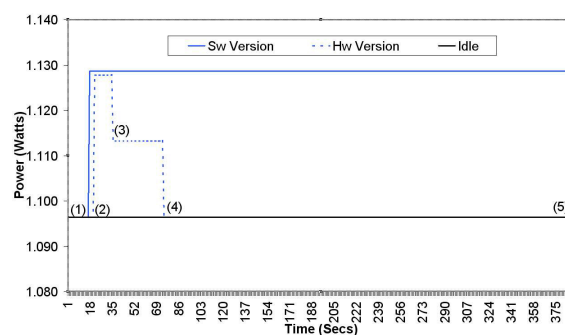


Figure 8: Suzaku power consumption diagram

8.2 The sensor monitor application

The monitor client is used to start / stop the sensing activity of the server and to display the values received. The server starts in an idle state. When it receives a start command, it launches a child process that requests the addition of a sensor specific softcore module. If the request is not satisfied the process terminates and the server sends back a failure message. Else, the child process starts reading sensor values and forwards them to the client. The child process can be terminated at any point in time via a corresponding client command.

The softcore device requested by the child process cannot function properly without a corresponding sensor being attached to the Suzaku board. For this reason the addition request is not considered unless the appropriate sensor is connected (by hand) to the board. When the sensor is disconnected from the board, the child process

receives a driver error and terminates, making it possible for the corresponding softcore module to be removed.

8.3 Configuration scenarios

We have configured the bitstream server to deliver four bitstream files that have been pre-built for this particular setup: (1) the base system configuration, (2) the base configuration plus the mandelbrot accelerator module, (3) the base configuration plus the sensor access module, and (4) the base configuration plus the mandelbrot accelerator and the sensor access modules.

The system is started with the first configuration. From that point onwards, any other configuration can be dynamically installed, depending on the sequence of requests issued by the mandelbrot and monitor applications (via their respective clients) as well as the physical presence of the sensor. Given that reconfiguration does not take place solely for the purpose of device removal, the system will stop reconfiguring once configuration (4) has been installed, because in this case all (future) requests issued by these applications are trivially satisfied.

9 Conclusion

In this paper we have presented the design and implementation of system-level mechanisms and application-level support for the dynamic addition and removal of softcore devices on a reconfigurable SoC featuring a softcore CPU and embedded operating system. This functionality is achieved without relying on DPR. Although the entire FPGA is re-programmed from scratch when a reconfiguration takes place, system, application and relevant device state can be maintained to a large degree, thereby achieving satisfactory transparency.

Application programming support comes in the form of a library for issuing device addition/removal requests that are asynchronously acknowledged via signals. Reconfiguration is triggered via a user-level process that collects and handles application requests in a bundled fashion. The configuration bitstream is downloaded from a remote server over the network, making it possible to support resource constrained systems with communication capability. In case of resource scarcity, priority is given to critical devices. Once the bitstream is saved in the designated memory area for programming the FPGA, reconfiguration (during which application processes remain frozen) takes less than a second to complete in our current platform.

Finally, we have considered softcore devices that rely on off-chip peripherals, and have extended our implementation to take such device addition requests into account only if the required peripheral is physically connected to the system.

10 Acknowledgments

This work was supported by the Greek General Secretariat for Research and Technology through the 3rd Community Support Programme; Measure 8.3: Research and Technology Human Resources; Action 8.3.1: “Reinforcement Programme of Human Research Manpower - PENED2003”.

11 Availability

More information about this work and uClinux patches are all available at <http://www.inf.uth.gr/vss>

References

- [1] ATMARK TECHNO INC. *Suzaku Series*. <http://www.atmark-techno.com/en/products/suzaku>.
- [2] ATMEL, I. *Field Programmable System Level Integrated Circuits (FPSLIC)*(2002). <http://www.atmel.com/products/FPSLIC/>.
- [3] BLODGET, B., JAMES-ROXB, P., KELLER, E., McMILLAN, S., AND SUNDARARAJAN, P. A Self-Reconfiguring Platform. In *Proceedings of Field Programmable Logic and Applications* (2003), pp. 565–574.
- [4] CUNNINGHAM, N. *Suspend2 project*. <http://www.suspend2.net/>.
- [5] DYER, M., PLESSL, C., AND PLATZNER, M. Partially reconfigurable cores for xilinx virtex. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications* (London, UK, 2002), Springer-Verlag, pp. 292–301.
- [6] FLETCHER, B. FPGA Embedded Processors: Revealing True System Performance. In *Embedded Systems Conference San Francisco* (2005), no. ETP-367.
- [7] FONG, R. J., HARPER, S. J., AND ATHANAS, P. M. A versatile framework for fpga field updates: An application of partial self-reconfiguration. In *IEEE International Workshop on Rapid System Prototyping* (2003), pp. 117–123.
- [8] HAUBELT, C., KOCH, D., AND TEICH, J. Basic OS Support for Distributed Reconfigurable Hardware. In *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation* (Samos, Greece, July 2003), pp. 18–22.
- [9] HORTA, E. L., AND LOCKWOOD, J. W. Parbit: A tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (fpgas). Tech. Rep. WUCS-01-13, Washington University Department of Computer Science, 2001.
- [10] LU Y, BERGMANN N, W. J. Dynamic loading of peripherals on reconfigurable system-on-chip. In *SPIE Microelectronics: Design, Technology, and Packaging II* (2005), vol. 6035.
- [11] MATSUMOTO, Y., AND MASAKI, A. Speed improvement of FPGA by mixing multiple-gate-width routing switches. In *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)* (2005), pp. 14–22.
- [12] NOLLET, V., MIGNOLET, J.-Y., BARTIC, A., VERKEST, D., VERNALDE, S., AND LAUWEREINS, R. Hierarchical run-time reconfiguration managed by an operating system for reconfigurable systems. In *Engineering of Reconfigurable Systems and Algorithms* (2003), pp. 81–87.

- [13] RAGHAVAN, A. K., AND SUTTON, P. Jpg - a partial bitstream generation tool to support partial reconfiguration in virtex fpgas. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2002), IEEE Computer Society, p. 192.
- [14] SHANNON, L., AND CHOW, P. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays* (2004), pp. 190–199.
- [15] SIDHU, R. P. S., AND PRASANNA, V. K. Efficient metacomputation using self-reconfiguration. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications* (London, UK, 2002), Springer-Verlag, pp. 698–709.
- [16] STITT, G., VAHID, F., MCGREGOR, G., AND EINLOTH, B. Hardware/software partitioning of software binaries: a case study of h.264 decode. In *CODES+ISSS* (2005), pp. 285–290.
- [17] UNGERER, G. netflash utility. <http://docs.linux.com>, Oct. 2002.
- [18] VULETIC, M., POZZI, L., AND IENNE, P. Programming transparency and portable hardware interfacing: Towards general-purpose reconfigurable computing. In *ASAP* (2004), pp. 339–351.
- [19] WILLIAMS, J. *The Microblaze-uClinux kernel port Project*. <http://www.itee.uq.edu.au/jwilliams/mblaze-uclinux/>.
- [20] WILLIAMS, J., AND BERGMANN, N. Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms* (2004), pp. 163–169.
- [21] XILINX INC. *Architecting Systems for Upgradability with IRL*, 2001. Application Note XAPP412.
- [22] XILINX INC. *Two Flows for Partial Reconfiguration: Module Based or Difference Based.*, 2003. Application Note XAPP290.

Resilient Connections for SSH and TLS

Teemu Koponen
Helsinki Institute for Information Technology
teemu.koponen@hiit.fi

Pasi Eronen
Nokia Research Center
pasi.eronen@nokia.com

Mikko Särelä
Helsinki University of Technology
Laboratory for Theoretical Computer Science
id@tcs.hut.fi

Abstract

Disconnection of an SSH shell or a secure application session due to network outages or travel is a familiar problem to many Internet users today. In this paper, we extend the SSH and TLS protocols to support resilient connections that can span several sequential TCP connections. The extensions allow sessions to survive both changes in IP addresses and long periods of disconnection. Our design emphasizes deployability in real-world environments, and addresses many of the challenges identified in previous work, including assumptions made about network middleboxes such as firewalls and NATs. We have also implemented the extensions in the OpenSSH and PureTLS software packages and tested them in practice.

1 Introduction

An increasing number of Internet hosts are mobile and equipped with more than one network interface. Simultaneously, operation of mobile hosts has become more continuous: the hosts have long uptimes, and the applications do not need to be closed when the host enters a “suspended” state. However, in the today’s Internet, applications experience this combination of improved connectivity and operation as a less stable networking environment. This is mainly because transport layer connections break more frequently due to changes in IP addresses, network failures, and timeouts during disconnected or suspended operation.

It is often desirable to hide these disruptions from the end user. For instance, a user should be able to suspend a laptop, move to a different location, bring up the laptop, and continue using the applications that were left open with minimal inconvenience. In other words, the system should provide *session continuity* over the disruptions in network connectivity (cf. Snoeren’s analysis of the session abstraction [23]).

Traditionally, session continuity has been considered as a part of mobility, and has been handled in the data link layer (e.g., wireless LAN or GPRS handover mechanisms) or in the network layer (e.g., Mobile IP). However, there are a number of reasons why providing session continuity higher in the protocol stack is desirable:

Long disconnection periods: while network-layer mobility mechanisms can deal with changing IP addresses, they cannot help the transport layer to overcome likely timeouts during long disconnections. Moreover, how exactly should long disconnections be handled often depends on the application in question.

No network infrastructure: in today’s Internet it is common that clients are mobile but servers are not. In this kind of environment, session continuity can be provided without requiring the deployment of additional fixed infrastructure (such as Mobile IP home agents).

Applications get upgraded: it is often claimed that mobility has to be low in the stack to enable it for a large number of different applications. However, we hypothesize that it is often actually easier to deploy resilient mechanisms built into applications. After all, the applications get upgraded all the time and processes for that exist; but installing and configuring a Mobile IP implementation is beyond capabilities of most users and system administrators.

Limited end-to-end connectivity: mobility mechanisms implemented in the network or transport layer may not work across various types of middleboxes that are present in the network. For instance, if a firewall near a client allows only outbound TCP connections, Mobile IP does not work. Session continuity mechanisms integrated into applications make the least number of assumptions about the network between the endpoints.

These arguments suggest that the session layer is the lowest layer to implement *resilient connections* that can span several sequential transport layer (TCP) connections, and thus, survive not only changes in IP addresses, but also relatively long periods of disconnection.

In this paper, we extend two common secure session layer protocols to support resilient connections: Secure SHell (SSH) Transport Layer Protocol [28, 29] and Transport Layer Security (TLS) [3].¹ We have implemented these extensions in two open-source software packages: OpenSSH, the most popular SSH implementation [16], and PureTLS, a Java TLS library [20].

Our main contributions are as follows. First, we have developed resiliency extensions for the common TLS and SSH protocols that largely avoid the deployability problems associated with previous proposals. Second, we have analyzed the challenges faced when implementing this kind of extensions to legacy software packages that were not designed with resiliency in mind. In particular, different styles of handling concurrency and I/O have large implications for the implementations: OpenSSH uses asynchronous (select-based) I/O with a process for each client, while PureTLS uses synchronous I/O with threads.

The rest of the paper is structured as follows. In Section 2, we introduce the SSH and TLS protocols and previous work on resilient connections. Our design principles, described in Section 3, attempt to address deployment challenges we have identified in the existing proposals. In Section 4, we introduce our extensions to the SSH and TLS protocols. Section 5 describes our prototype implementations, which are then evaluated in Section 6. Finally, Section 7 summarizes our conclusions and discusses remaining open issues.

2 Background and related work

The Secure Shell (SSH) is a protocol for secure login and other network services [28]. It consists of three main sub-protocols: the SSH transport layer protocol, user authentication protocol, and connection protocol. The SSH transport layer protocol is the lowest layer, and is responsible for authenticating the server and providing an encrypted and integrity-protected channel for the other sub-protocols. The user authentication protocol authenticates the client, while the connection protocol multiplexes several logical connections (such as interactive terminal sessions, X11 window system forwarding, and TCP/IP port forwarding) over a single transport layer connection.

Transport Layer Security (TLS) is a session layer protocol providing encrypted and authenticated communication session for communication between two applications [3]. It consists of two major parts: the TLS record protocol provides a secure communication channel to upper layers, and is responsible for encryption and integrity protection of data. The TLS handshake protocol provides

¹Note that despite their names, both protocols are strictly above the transport layer (TCP) in the protocol stack, and thus calling them session-layer protocols is more accurate.

the key material and authentication for the TLS record protocol; this usually involves X.509 certificates and a key exchange based on RSA encryption. The two remaining components of TLS, the alert and change cipher spec protocols, are beyond the scope of this paper.

The benefits of providing session continuity above the transport layer have been recognized before; for instance, Duchamp [4] and Snoeren [24] provide several arguments in its favor. There is a large number of proposals that provide resilient connections above the transport layer but below the application layer protocol: Persistent connections [32], Mobile TCP socket [18, 19], MobileSocket [15], SLM or Session Layer Mobility [11], Reliable sockets [30], Migrate [23], Robust TCP connections [5], NapletSocket [33], Channel-based connectivity management [26], and Dharma [13], to mention just a few examples.

The common part of most of these proposals is a library placed above the transport layer but below the sockets API used by the application. The library presents a single unbroken communication channel to the application, hiding transport layer disruptions from the applications. The library is responsible for the signaling required to manage the multiple TCP connections, and also buffers application data so it can be retransmitted over a new TCP connection if necessary (this is required since most operating systems do not allow applications to access the TCP buffers).

However, implementing resilient connections in the “sockets API” layer has a number of drawbacks.

- The proposals typically use out-of-band signaling: a separate TCP connection (or UDP-based “session”) coordinates multiple TCP connections. This can lead to deployment problems if, e.g., a firewall allows the port used by the application itself, but not the port used for resiliency signaling. An important reason for out-of-band signaling is the lack of an extension negotiation mechanism in the sockets API layer; however, such a mechanism is essential for incremental deployment. While some proposals (such as Zandy’s reliable sockets [30]) do actually implement the initial resiliency signaling in-band, they rely on obscure TCP semantics with questionable deployability properties. However, even these solutions change to out-of-band signaling after the connection setup (e.g., due to TCP’s head-of-line blocking issues).
- A separate key exchange is required to protect the signaling messages (if the messages are protected at all). This introduces additional overhead.
- While a separately delivered dynamically linked library that “hijacks” the operations of the normal

socket calls is a good approach for research, it creates deployment problems if it does not come bundled and tested with the software with which it is to be used. Deploying such separate component is likely to get less-than-enthusiastic response from, e.g., corporate IT departments who would have to deploy and manage this component in mission-critical environments.

Proposals to implement the session continuity even higher in the protocol stack than the session layer exist. If an application protocol connection setup is a lightweight operation (e.g., HTTP GET), it's not necessary to extend any protocol to implement reconnections. An application just reconnects and at the same time minimizes the visibility of the reconnection to its user. For example, most modern mail user agents operate in this manner.

An application protocol connection setup may consume an considerable amount of resources, however, and thus, several application protocols have been extended to provide session continuity. For instance, REX, an SSH-like remote execution utility [9], the XMOVE extension to the X Window System [25], the REST extension to FTP [6], and SIP [22] all allow continuing a session even if a transport layer connection is disrupted for some reason. These extensions are typically very specific to the application in question; in contrast, our TLS extensions would work with any application-layer protocol run over TLS.

Session continuity for interactive terminal sessions can also be provided by decoupling the terminal seen by applications from the remote terminal session, as done in, for instance, Screen [7] and Bellovin's Session Tty Manager [1]. However, these approaches still require the user to manually establish a new SSH connection and re-attach the terminal session.

Proposals that operate in the transport layer (e.g., Huitema's Multi-Homed TCP [8]) are beyond the scope of this paper. As they require modifications to the operating system's TCP/IP stack, and may not work with existing middleboxes, we do not consider them easily deployable.

3 Design principles

Based on the existing work, we have set our design principles to emphasize deployability.

No network changes: no extra requirements for the network or middleboxes between two communicating hosts should be set. As an example, the extensions must not require any additional configuration in firewalls.

Incremental deployment: the extensions should provide functionality once both connection end-points support it. The extensions should also interoperate with

legacy end-points without the extensions.

Limited end-point changes: the extensions should require only modifications in TLS and SSH implementations, but no operating system changes or additional software components. The latter includes, e.g., dynamic libraries interposed between the application and the operating system.

In terms of functionality, our design principles were:

Disconnections may last long: the extension should deal gracefully with long periods of disconnection. The maximum supported disconnection period is a local policy issue, and not a protocol issue. Thus, the protocol extensions should not limit the duration of disconnections.

No handover optimization: the extensions are not optimized for fast handovers. This is mainly because we believe the default disconnection to be relatively long (from tens of seconds to hours).

In addition to the protocol extensions, there are certain implementation aspects to be considered. Server side concurrency is the most important one. The mechanisms used to implement concurrency often depend on the operating system and programming language used. Obviously, our extensions should not prevent typical server implementation strategies such as "a process for each client" (either forked on demand or beforehand), "a thread for each client", or select-style asynchronous I/O.

4 Protocol extensions

In this section we describe the extensions made to the SSH and TLS protocols. Since the extensions have much in common, we present the shared features first, followed by the SSH and TLS specific details.

4.1 Common features

In-band signaling: deployability concerns in practice mandate the use of in-band signaling. In other words, information required by the extensions is sent as part of normal SSH and TLS messages, and all TCP connections are initiated by the client. This ensures that resilient connections do not introduce any additional requirements for the network between the client and the server.

Extension negotiation: incremental deployment requires interoperability with endpoints that do not support these extensions, and thus, their use has to be negotiated. Fortunately, both SSH and TLS have mechanisms for negotiating protocol features when the connection is set up.

Securing signaling: when the client creates a new TCP connection to the server, it has to somehow indicate that it wants to continue a previous session, and prove that it is indeed the same client as previously. Thus, we need a way to identify an existing session (any public and unique information exchanged during the session setup

will do) and way to authenticate the signaling. Since both SSH and TLS establish session keys between the client and the server, the authentication is relatively easy to do.

Buffer management: both SSH and TLS operate over TCP, which provides a reliable lossless connection channel. However, when the TCP connection breaks, the TCP socket buffers may contain data that was not yet received by the other endpoint, and thus, the data has to be retransmitted when a new TCP connection is created. Since operating systems typically do not allow access to the TCP buffers, separate buffers have to be maintained in the application.

Previously, two different approaches have been used for managing these buffers: either data is removed from the buffer only when an explicit session layer acknowledgement is received (e.g., MobileSocket by Okoshi et al. [15]), or the buffer size is limited to the size of TCP buffers (e.g., Zandy’s reliable sockets [31]). We chose the former approach: the endpoints send acknowledgements regularly (say, after receiving 64 kB from the peer). While the nodes may know their own TCP buffer sizes, the network may also contain transport layer proxies that buffer data: for instance, TLS is often run through web proxies using the “CONNECT” method [12]. Thus, while explicit acknowledgements add some overhead, only they ensure that the extensions work properly in existing network environments. This corresponds to the “end-to-end argument” by Saltzer et al. [21]: since parts of TCP may be implemented by the communication system itself, end-to-end reliability can be correctly implemented only above TCP.

Closing: SSH and TLS connections are both tightly bound to an underlying TCP connection. The resiliency extensions render the situation more complex: if the TCP connection breaks, the server should wait for the client to reconnect again. Thus, the protocol should have an explicit “close” message to be used when the endpoints actually want to close the session permanently. Fortunately, both the SSH transport layer protocol and TLS have this kind of messages. However, we discovered that OpenSSH did not actually send the close message, since previously there was no need to differentiate between a gracefully closed session and a broken TCP connection.

4.2 Extending SSH

Resilient connections for SSH could be implemented either in the SSH transport layer protocol or the connection protocol. In the end, we decided to implement our extension in the SSH transport layer protocol, since this seemed to be simpler, and had more in common with the TLS extensions described in the next section.

The SSH protocol suite is extensible: in the transport layer protocol, the client and the server negotiate the al-

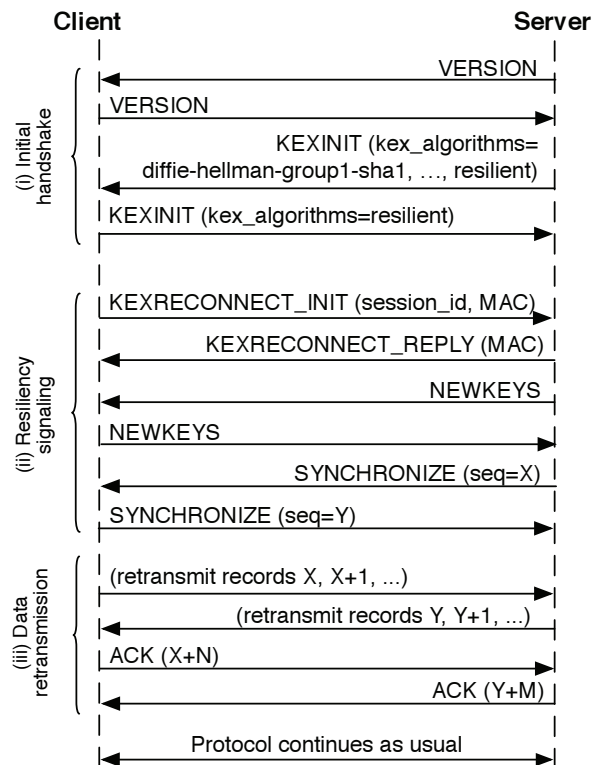


Figure 1: Reconnecting an existing SSH session.

gorithms that will be used during the session. However, while the algorithms are negotiable, the negotiation does fix algorithm categories. Thus, we had to re-use an existing category to negotiate the resiliency support: the client and the server announce their support for this extension by including a Key EXchange (KEX) algorithm named “resilient” as the least-preferred algorithm. If both endpoints supports this extension, they enable buffering of data and sending of explicit acknowledgements. The acknowledgement is a new SSH message type that contains the sequence number of the next expected record.

Modeling the resiliency extension as a special key exchange algorithm also simplifies things when the client wants to reconnect; i.e., continue the same session over a different TCP connection. The exchange is shown in Figure 1. The client indicates that it wants to continue a session by listing “resilient” as the only supported key exchange algorithm. The client then sends a message containing a session identifier and a Message Authentication Code (MAC); the server responds with its own MAC. The MACs prove that the parties are still the same as in the original connection, and are calculated over the VERSION and KEXINIT messages (which include nonces to prevent replays). The MAC is calculated using a separate key used only for the KEXRECONNECT messages, and is derived during the initial handshake at the same time

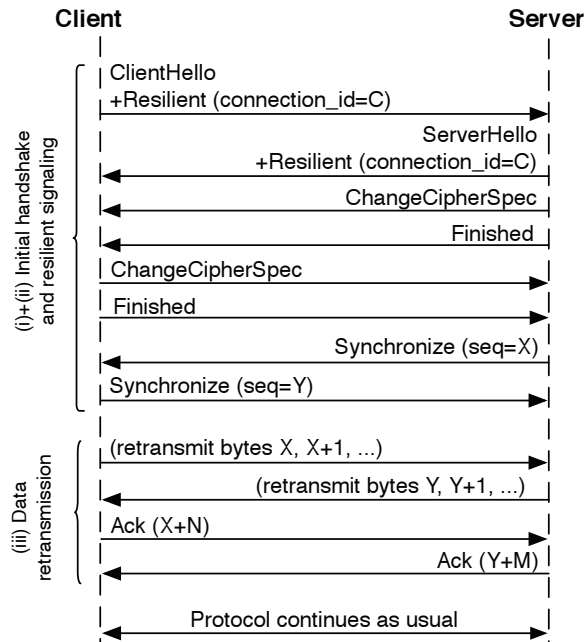


Figure 2: Reconnection procedure in TLS.

as the encryption and integrity protection keys.

After this, the endpoints take the cryptographic keys into use, send a “synchronize” message indicating where the previous connection was broken, and retransmit lost data from the buffers.

The SSH transport layer supports payload compression. While the transport layer protocol implements the compression, compression is done before encryption. Thus, the compression belongs to the topmost part of the transport layer. We decided to hide the connection disruptions from the compression engine to maintain the compression engine’s state intact. Re-establishing the compression state would only decrease the compression performance during reconnections.

4.3 Extending TLS

The resiliency extension to TLS is negotiated using the TLS extension mechanism [2] in ClientHello/ServerHello messages. Similarly as in the SSH case, if both endpoints support this extension, they start buffering data and sending acknowledgement messages. In the TLS case, the acknowledgement messages contain the number of application data bytes received instead of the TLS record sequence numbers. We chose this approach since the reconnection handshake is based on the abbreviated TLS handshake which resets the sequence numbers back to zero.

The reconnection exchange is shown in Figure 2. The client indicates that it wants to continue an existing

connection by including a connection identifier in the ClientHello message. The ClientHello/ServerHello messages are followed by an abbreviated handshake (based on the normal TLS session resumption handshake) which verifies that the parties have remained the same and establishes fresh session keys.

After this, the endpoints tell how much data needs to be retransmitted, and retransmit the lost data, if any.

It is important to note that while the cryptographic handshake re-uses an existing TLS feature called “session resumption”, there is an important difference. TLS session resumption is a feature of the TLS handshake protocol which caches the results of expensive public-key operations. It is a performance optimization and is independent of the actual data transfer (the TLS record protocol). Thus, it does not enable a client to continue an existing connection that was for some reason broken.

4.4 Security analysis

Making SSH and TLS sessions resilient to disconnections could introduce new security vulnerabilities. However, we believe that the extensions presented in this paper provide the same level of security as the situation when new SSH and TLS sessions are initiated to handle disconnections. In this section, we provide a high-level analysis of our protocol extensions. A complete security analysis of our protocol is beyond the scope of this paper.

In our extensions, all messages are authenticated using shared keys created during the initial SSH or TLS protocol exchange. Thus, an attacker cannot spoof or modify the reconnect messages. Replay attacks are not possible, since the first SSH and TLS key exchange messages include fresh nonces that are covered by a MAC later during the handshake.

Since the extensions require the endpoints to buffer data that has not been acknowledged, the amount of resources needed by a single SSH or TLS session is increased. Thus, the work required for a denial of service attack against a server (by creating a large number of sessions) may be less than in normal SSH or TLS. However, in most cases the buffers are likely to represent only a small share of the resources, and thus, denial of service resistance is not significantly changed.

5 Implementation considerations

In this section, we analyze the implications of resilient connections for SSH/TLS client and server side implementations.

5.1 When to reconnect and which interface to use?

Resiliency against connection disruptions brings a new challenge to client and server side implementations of both protocols. On the client side, the challenge is to determine when to start the reconnection procedure. Some options include the following:

1. A manual request; e.g., a user could click a “reconnect now” button in the application user interface.
2. Automatically when the device is brought up from a “suspended” power management state.
3. Whenever the current TCP connection is broken.
4. Whenever a more preferred network interface is available.
5. Probably several more options exist.

In addition to deciding when to reconnect, there may be multiple interfaces available: which of these should be used to establish the connection?

To ensure easy deployability, the solution should depend only on tools and APIs commonly available on the deployment environment and not require any additional software on the client machine.

Therefore, we decided to simply rely on the operating system’s source address selection. In other words, we leave it to the operating system to decide which local interface should be used when a TCP connection is established, and initiate reconnection when the operating system’s decision changes, or the current TCP connection is broken.

This raises the question of how to notice that the OS’s source address selection policy has changed. In Windows, the Winsock API has a feature (“SIO_ROUTING_INTERFACE_CHANGE” socket option; see [14]) that allows the application to be notified of changes. BSD-based Unixes have “PF_ROUTE” routing sockets [27] and Linux has Netlink sockets [10] that also allow monitoring of routing table changes.

In the end, we implemented two different approaches. For OpenSSH, we used a routing socket to monitor routing table changes. In PureTLS, we settled for polling the OS in regular intervals to see if the preferred interface has changed. The polling can be done, for instance, by creating a “connection-mode” UDP socket, and reading the local address using the `getsockname()` API call (note that no UDP packets are actually sent). The approach was preferable in Java, since it avoided the need to have native and platform-specific code.

On the server side, a certain level of uncertainty is imminent too. For a server, the challenge is to determine

the time to discard a session that waits for its client to reconnect. The difference to the client side is that the server must make the decision completely without the help of a user. Our vision is that the time a server is willing to keep resources allocated for a session without a connected client is a local policy issue. Different users may have different timeouts as well as servers with different loads may have different timeouts. For instance, one could assume a shared server is willing to maintain sessions shorter period of time than a server solely used by a single user. For the prototype implementations, we implemented a configurable server-wide timeout.

5.2 Server side concurrency

A common server design strategy is to create a new process or thread for each new client connection. While this often simplifies the server design, in this context concurrency becomes a complicating factor, since it results in a situation where the client’s original session and reconnection request are handled by two different processes or threads.

A server designer has two options to choose from: either the new process finds the corresponding old process and passes the new TCP connection to the old process, or the other way around. Regardless of the choice, the server must maintain a table mapping sessions to processes for inter-process (or inter-thread) communication. In our implementations, the new process passes the new connection to the old process. Before passing the connection, the new process validates reconnection attempts. The validation requires contacting the old process, as the new process has no other access to the session keys. Once the new process has passed the connection to the old process, it exits.

Two reasons made us to choose the new process/thread to pass its state to the old process/thread. First, the new process has simply less state to pass: in practice, passing a file descriptor of a transport connection and sequence numbers to synchronize is sufficient. Second, besides the amount of state, the new process has state information that is easier to transfer. The old process can have such state that is impossible to pass across process boundaries. As an example, consider a TLS server process that creates child processes. In majority of platforms, it is impossible to pass child processes from a process to another—which would be a requirement if the old process passed its state to the new process.

In a multi-threaded server, implementing state passing is straightforward. However, if a server is implemented using concurrent processes, the above indicates that the server requires certain Inter-Process Communication (IPC) facilities:

1. An inter-process message channel to validate recon-

nection requests using keys stored in the old process.

2. An inter-process message channel to pass sequence numbers for synchronization.
3. A file descriptor passing mechanism to transfer a transport connection (socket) from the new process to the old process.

The required IPC facilities are realistic on most modern platforms, but they have often platform-specific features. Therefore, while the resiliency extensions are unlikely to prevent porting a server implementation to another platform, IPC mechanisms may add an extra twist to the porting process.

5.3 Atomic reconnections

Reconnection attempts must be atomic: the protocol state machine of an old connection must not become corrupted if an attempt fails. As discussed above, we designed the server implementations in a way that the new process transfers its state to the old process only after a reconnection request is determined to be valid. The approach has a positive side effect: the server side reconnection handling becomes atomic from the old process' point of view. If a reconnection request is invalid, the old process sees nothing.

Client implementations required similar atomic reconnection attempts: either a reconnection attempt succeeds or no state is affected. Unfortunately, implementing this in a general case can be challenging, as we learned in a hard way. A normal client implementation can modify global variables and data structures while connecting, and if connecting fails, it simply exits. Being a perfectly valid approach without resiliency extensions, this becomes challenging when the client implementation should behave in a deterministic manner in the case of connection failures.

Our observation was that it is tempting to modify a client implementation to behave as a multi-process or multi-threaded server: a fresh client process or thread attempts to reconnect and only once it succeeds, it passes its state to the old process or thread. In this way, the client implementation may dirty the new process or thread state but the attempt still remains atomic from the old process' point of view. As we implemented our clients in this way, we found out an unfortunate side effect: clients implemented in a process model require similar IPC facilities as the servers do. Our OpenSSH client was implemented as a multi-process and PureTLS client as multi-threaded client.

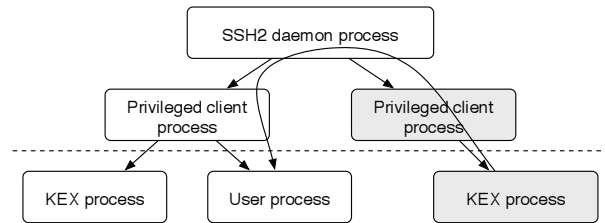


Figure 3: OpenSSH separates privileged processes (above dashed line) and less privileged processes (below dashed line). Grey boxes depict the processes processing a reconnection request.

5.4 Interface to higher layers

SSH transport layer protocol and TLS are not useful alone; they are always used together with some higher layer protocol. TLS is used with many different applications, while in the SSH case, the higher layer protocols are the SSH connection and user authentication protocols.

In general, we would like to change the interface offered by TLS and SSH transport layer protocol as little as possible. However, some changes and/or enhancements may be desirable. For instance, in the TLS case, some applications may be interested in knowing when a connection is no longer working, when a reconnection has happened, or even initiating reconnection.

Another set of issues arises from the fact that the IP addresses and port numbers used by the TCP connections may change when reconnecting. If the application uses these values for some other purpose than just sending packets, it may want to know when they change. For instance, OpenSSH can be configured to allow connections only from certain IP addresses. Similarly, a Java application can retrieve the addresses using Socket object methods such as `getInetAddress()`, and use them for, e.g., access control. Thus, it would be useful to have callbacks that allow the application logic to be notified when the addresses change.

These changes in the higher layer interfaces may require small modifications to OpenSSH and applications that use PureTLS. However, we have not yet implemented or further explored these modifications in the current versions of our prototypes.

5.5 OpenSSH

OpenSSH implements privilege separation to limit the effects of possible programming errors [17]. In the privilege separation, a privileged server daemon process uses less privileged processes to interface with clients. Less privileged processes then communicate with the privileged process through a monitor that protects the priv-

ileged process. Figure 3 depicts how OpenSSH forks (straight arrows) a separate process to do the key exchange and user authentication. Once the KEX process is done, the OpenSSH server forks yet another process to actually serve the client. Only the last process runs under the user's identity.

The OpenSSH privilege separation requires state serialization and passing across process boundaries: different processes perform the key exchange and the actual connection serving. After the key exchange, the KEX process serializes its key material together with information about the agreed algorithms and passes the state to its privileged parent process. The parent process then forks the actual connection serving process and passes the state further there.

It turned out that for both the OpenSSH server and client implementations, privilege separation facilities based on Unix socket pairs were enough to provide the required IPC facilities once they were extended to pass file descriptors. No additional authentication between processes was necessary either; the Unix socket pairs are invisible beyond a process and its child processes. On the server side, facilities provide atomic reconnections and the transfer of a new connection to an old process. On the client side, they only guarantee atomic reconnections as discussed earlier.

On the server side, we decided to transform the main daemon process into a message broker as it was the only common factor between all processes. The curved arrow in Figure 3 depicts how a connection together with synchronization information actually travels through several processes via the main daemon process, from the new process eventually to the old process.

5.6 PureTLS

In PureTLS, most of the implementation complexity comes from the requirement to keep the objects visible to the application (such as Socket, InputStream and OutputStream instances) unchanged over reconnections.

For Socket, this required creating an additional layer of indirection: a new Socket instance that forwards the method calls to the "real" underlying socket. Fortunately, PureTLS already contained this kind of indirection layer, and only small modifications were needed to allow changing of the underlying socket on-the-fly.

6 Evaluation

In this section we present measurement results of reconnection transactions for both protocols and discuss the complexity of implementations. In this paper, we did not focus on the performance optimizations. Instead, the

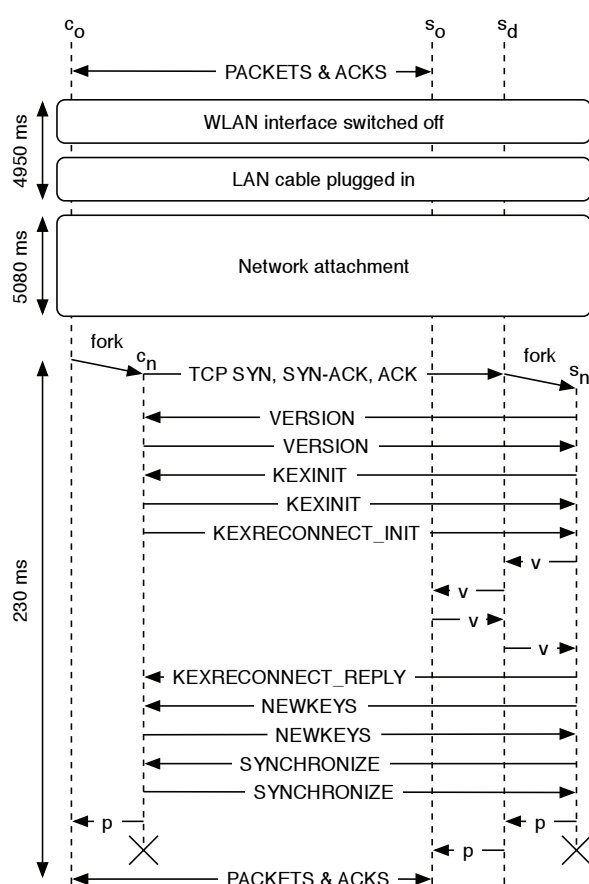


Figure 4: SSH processes involved in reconnecting a session.

main purpose of our evaluation is to show that our prototypes work and their performance is adequate for the intended use (relatively long disconnections).

6.1 Measurements

One of our main assumptions behind the design principles was that typical connection disruptions last a relatively long time. Therefore, we constructed one such scenario: a user manually switches from Wireless LAN to wired Ethernet. While the access to Internet from both networks goes through different NAT boxes, the user still expects his connections to survive from a changing IP address and NAT box. We conducted a set of measurements to validate the hypothesis and measure the actual expected length of typical reconnections.

In our scenario, the user downloads a large file from a remote server, either over SFTP or TLS. First, a user's laptop is attached to a wireless access point, but then the user decides to connect it to a fixed LAN to access remote services not available through the restricted public WLAN. Switching the access point requires, besides

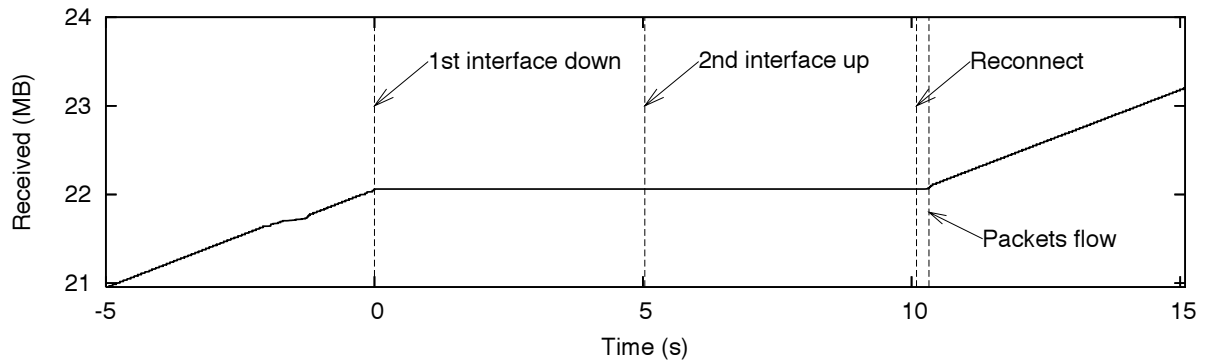


Figure 5: Progress of an SFTP transfer before, during, and after reconnection.

plugging an Ethernet cable, also turning off the WLAN interface; otherwise the laptop operating system keeps the WLAN interface as its primary interface.

We expect that typical disconnections would last significantly longer than the couple of seconds in these measurements. For example, we have used the extensions to keep SSH sessions alive while a suspended laptop is carried from the office to home.

OpenSSH measurements

In the OpenSSH tests, the remote download server was on the Internet and the round-trip time to the server was 10 ms through both WLAN and LAN. The SFTP client run on Mac OS X 10.4, while the SSH server was run on Linux.

As described in Section 5.1, the client can start reconnection not only when the TCP connection breaks, but also when the preferred source address has changed. Our OpenSSH extension uses a separate process to monitor the routing tables of the operating system. Once this process realizes that the route to the server has changed, it sends a signal to other processes that handle the actual reconnection.

Figure 4 represents the reconnection from a process viewpoint. On the client and server sides, temporary processes (c_n and s_n , respectively) handle the reconnection and old processes (c_o and s_o) receive the new transport connection only when it is time to resend lost packets. The main daemon process (s_d) only brokers messages between processes. In the figure, arrows titled as 'v' depict the inter-process validation messaging, while 'p' arrows depict the actual state passing.

Figure 5 shows the number of bytes an SFTP client has received as a function of time. The test user turned off the WLAN interface at time zero. The user quickly plugged a wired LAN cable in; finding the cable and inserting it to the laptop took less than three seconds. While it did not take that long to request an IP address (in the figure interface is up once it has an IP address), the graph illustrates how long it actually took before the network

attachment was completely over from the SFTP point of view. Before the SFTP client receives a signal from the routing table monitoring daemon, 5 seconds has passed since the wired LAN interface came up. The actual reconnection then takes only about 200 ms before the file download continues.

PureTLS measurements

In the PureTLS tests, the client run on Linux and the server on Windows XP; the round-trip time to the server was around 1 ms.

Figure 6 shows the number of bytes received as a function of time. In this case, the network disruption lasted 5.5 seconds, and recovering from it took about 0.5 seconds. The differences compared to the OpenSSH case are explained mainly by how the reconnection is triggered (see Section 5.1).

Acknowledgment overhead

Figures 5 and 6 show only the downlink traffic and do not contain the additional network traffic caused by the session layer acknowledgements. However, this traffic is tolerable, and does not necessarily generate additional IP packets since the SSH/TLS ACKs can fit in the same packets as TCP ACKs.

Our OpenSSH acknowledgment implementation was suboptimal, since it acknowledges every received SSH transport layer message. While a single SSH ACK payload consumed only 5 bytes of space (packet type and 32-bit sequence number), the minimum cipher block sizes and MAC together increased the total size of ACK messages; a single ACK, with default OpenSSH configuration, consumed 32 bytes in total. Despite this suboptimal implementation, the extra traffic caused by the ACKs was more than acceptable, since the SSH transport layer messages can be up to 32 kilobytes: the ACK traffic amounted to less than 0.6% of the whole bandwidth. The PureTLS implementation does not acknowledge all records, but instead attempts to send ACKs at the same time as application data; the overhead figures were comparable to the OpenSSH case.

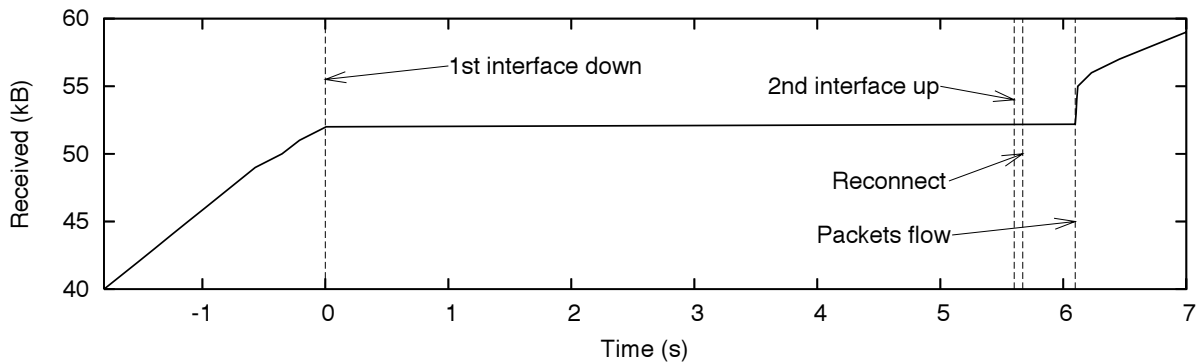


Figure 6: Progress of an TLS transfer before, during, and after reconnection. The temporary rapid speed-up after reconnection is caused by an implementation anomaly: the file transfer application uses a significantly smaller block size than the retransmissions done by PureTLS.

6.2 Implementation complexity

In-band resiliency signaling helps deployment, but it has an obvious extra cost: server and client applications must be modified. Next, we will briefly discuss the required implementation effort in the light of experiences from OpenSSH and PureTLS.

Our OpenSSH extension required about 2,200 lines of code. Over half of this code is related to passing state information and socket handles between the different processes. On the other hand, implementing explicit acknowledgments and buffering required relatively little effort, since much of existing OpenSSH functions were reusable as such. The OpenSSH implementation required roughly a man month of efficient work time, which included one complete refactoring.

The PureTLS extension was slightly simpler (about 1,000 lines of code) since Java's inter-thread communication facilities were much easier to use.

7 Conclusions

True *session continuity* in today's Internet requires not only handovers, but also gracefully handling long periods of disconnected operation. The contribution of our paper is three-fold. First, we have identified design principles that emphasize *deployability* and address some of the challenges in previous work. Second, following these principles, we have extended the SSH and TLS protocols to support resilient connections. Third, we have analyzed implementation issues faced when adding the functionality into two existing software packages.

Our three design principles are as follows: mechanisms for providing session continuity (a) should not place additional requirements for the network, (b) must allow incremental deployment, both providing benefits to early adopters and interoperating with legacy endpoints,

and (c) should not require changes in operating system or third party libraries.

Our experience with the SSH and TLS extensions indicates that these design principles mandate certain protocol features, the most important one being in-band signaling. Furthermore, the protocol needs to support extension negotiation and explicit close messages, and has to be extended with explicit acknowledgements for transferred data.

The required extensions to the TLS and SSH protocols were relatively simple. In our case, we embedded the resiliency negotiation into the initial connection setup messages in a backwards compatible manner. In addition, both protocols execute mutual authentication while reconnecting simply by proving the possession of the shared secret of a suspended session.

In the implementations, handling the server side concurrency was clearly the most challenging part. The process (or thread) that is handling the reconnection request must find the corresponding old process, since only the old process can validate the request. After a successful validation, the new process must pass the connection state and the TCP socket to the old process. This translates into inter-process or inter-thread communication mechanisms. Moreover, while dividing functionalities between the new and old processes, we found out that a new process should prepare a reconnection attempt and only alter the state of the old process after the reconnection attempt has succeeded. This simplified the implementations considerably.

While our paper has focused on addressing deployment challenges, deployability remains a difficult concept. Much of existing work on mobility has focused on issues easy to measure and compare, such as handover performance. Deployability in general, as well as approaches to compare it, have received less attention, and clearly, more work is needed to better understand how

different protocol design choices affect deployability.

Acknowledgments

The authors would like to thank N. Asokan, Dan Forsberg, Andrei Gurtov, Tobias Heer, Janne Lindqvist, and Pekka Nikander for their comments and suggestions. Helpful comments were also provided by Tero Kivinen who said he had planned a similar extension to SSH several years ago (however, no additional information is available about this work). Finally, we thank the anonymous reviewers and our shepherd, Stefan Saroiu, for their insightful comments.

References

- [1] Steven M. Bellovin. The “Session Tty” Manager. In *Proceedings of the Summer 1988 USENIX Conference*, pages 339–354, San Francisco, June 1988.
- [2] Simon Blake-Wilson, Magnus Nystrom, David Hopwood, Jan Mikkelsen, and Tim Wright. Transport Layer Security (TLS) Extensions. RFC 3546, IETF, June 2003.
- [3] Tim Dierks and Christopher Allen. The TLS protocol version 1.0. RFC 2246, IETF, January 1999.
- [4] Daniel Duchamp. The discrete Internet and what to do about it. In *2nd New York Metro Area Networking Workshop*, New York, NY, September 2002.
- [5] Richard Ekwall, Péter Urbán, and André Schiper. Robust TCP connections for fault tolerant computing. *Journal of Information Science and Engineering*, 19(3):503–516, May 2003.
- [6] Robert Elz and Paul Hethmon. Extensions to FTP. Work in progress (IETF Internet-Draft, draft-ietf-ftptext-mlst-16), September 2002.
- [7] GNU Screen. <http://www.gnu.org/software/screen/>, 2006.
- [8] Christian Huitema. Multi-homed TCP. Work in progress (draft-huitema-multi-homed-01), May 1995.
- [9] Michael Kaminsky, Eric Peterson, Daniel B. Giffin, Kevin Fu, David Mazières, and M. Frans Kaashoek. REX: Secure, Extensible Remote Execution. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, June–July 2004.
- [10] Andi Kleen et al. rnetlink, NETLINK_ROUTE – Linux IPv4 routing socket. Linux Programmer’s Manual, man page rnetlink(7), 2000.
- [11] Björn Landfeldt, Tomas Larsson, Yuri Ismailov, and Aruna Seneviratne. SLM, a framework for session layer mobility management. In *Proceedings of the 8th International Conference on Computer Communications and Networks (ICCCN ’99)*, Boston, MA, October 1999.
- [12] Ari Luotonen. Tunneling TCP based protocols through Web proxy servers. Work in progress (draft-luotonen-web-proxy-tunneling-01), August 1998.
- [13] Yun Mao, Björn Knutsson, Honghui Lu, and Jonathan M. Smith. Dharma: Distributed home agent for robust mobile access. In *Proceedings of IEEE INFOCOM 2005*, Miami, FL, March 2005.
- [14] Microsoft Corporation. Windows sockets 2. MSDN Library, http://msdn.microsoft.com/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp, 2006.
- [15] Tadashi Okoshi, Masahiro Mochizuki, Yoshito Tobe, and Hideyuki Tokuda. MobileSocket: Toward continuous operation for Java applications. In *Proceedings of the 8th International Conference on Computer Communications and Networks (ICCCN ’99)*, Boston, MA, October 1999.
- [16] OpenBSD project. OpenSSH. <http://www.openssh.org/>, 2006.
- [17] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003.
- [18] Xun Qu, Jeffrey Xu Yu, and Richard P. Brent. A mobile TCP socket. Technical Report TR-CS-97-08, Computer Sciences Laboratory, RSISE, The Australian National University, 1997.
- [19] Xun Qu, Jeffrey Xu Yu, and Richard P. Brent. A mobile TCP socket. In *Proceedings of the IASTED International Conference on Software Engineering*, San Francisco, CA, November 1997.
- [20] Eric Rescorla. Claymore PureTLS. <http://www.rtfm.com/puretls/>, 2006.
- [21] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
- [22] Henning Schulzrinne and Elin Wedlund. Application-layer mobility using SIP. *ACM SIGMOBILE Mobile Computing and Communications Review*, 4(3):47–57, July 2000.

- [23] Alex Snoeren. *A Session-Based Approach to Internet Mobility*. PhD thesis, Massachusetts Institute of Technology, 2002.
- [24] Alex C. Snoeren, Hari Balakrishnan, and M. Frans Kaashoek. Reconsidering Internet mobility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001.
- [25] Ethan Solomita, James Kempf, and Dan Duchamp. XMOVE: a pseudoserver for X window movement. *The X Resource*, 11, July 1994.
- [26] Jun-Zhao Sun, Jukka Riekk, Marko Jurmu, and Jaakko Sauvola. Channel-based connectivity management middleware for seamless integration of heterogeneous wireless networks. In *Proceedings of the the 2005 Symposium on Applications and the Internet (SAINT '05)*, Trento, Italy, January–February 2005.
- [27] Sun Microsystems. route – kernel packet forwarding database. Solaris 10 Reference Manual Collection, man page route(7P), 2003.
- [28] Tatu Ylönen and Chris Lonvick. The Secure Shell (SSH) protocol architecture. RFC 4251, IETF, January 2006.
- [29] Tatu Ylönen and Chris Lonvick. The Secure Shell (SSH) transport layer protocol. RFC 4253, IETF, January 2006.
- [30] Victor C. Zandy and Barton P. Miller. Reliable network connections. In *Proceedings of the 8th annual International Conference on Mobile Computing and Networking (MobiCom '02)*, Atlanta, GA, September 2002.
- [31] Victor Charles Zandy. *Application Mobility*. PhD thesis, University of Wisconsin-Madison, 2004.
- [32] Yongguang Zhang and Son Dao. A “persistent connection” model for mobile and distributed systems. In *Proceedings of the 4th International Conference on Computer Communications and Networks (ICCCN '95)*, Las Vegas, NV, September 1995.
- [33] Xiliang Zhong, Cheng-Zhong Xu, and Haiying Shen. A reliable and secure connection migration mechanism for mobile agents. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops (ICDCSW '04)*, Tokyo, Japan, March 2004.

Structured and unstructured overlays under the microscope

A measurement-based view of two P2P systems that people use

Yi Qiao and Fabián E. Bustamante

Department of Electrical Engineering & Computer Science, Northwestern University

{yqiao,fabianb}@cs.northwestern.edu

Abstract

Existing peer-to-peer systems rely on overlay network protocols for object storage and retrieval and message routing. These overlay protocols can be broadly classified as structured and unstructured – structured overlays impose constraints on the network topology for efficient object discovery, while unstructured overlays organize nodes in a random graph topology that is arguably more resilient to peer population transiency. There is an ongoing discussion on the pros and cons of both approaches. This paper contributes to the discussion a multiple-site, measurement-based study of two operational and widely-deployed file-sharing systems. The two protocols are evaluated in terms of resilience, message overhead, and query performance. We validate our findings and further extend our conclusions through detailed analysis and simulation experiments.

1 Introduction

Peer-to-peer Internet applications for data sharing have gained in popularity over the last few years to become one of today's main sources of Internet traffic [31, 12]. Their peer-to-peer approach has been proposed as the underlying model for a wide variety of applications, from storage systems and cooperative content distribution to Web caching and communication infrastructures. Existing peer-to-peer systems rely on overlay network protocols for object storage/retrieval and message routing. These overlay protocols can be classified broadly as either structured or unstructured based on the constraints imposed on how peers are organized and where stored objects are kept. The research community continues to debate the pros and cons of these alternative approaches [5]. *This paper contributes to this discussion the first multi-site, measurement based study of two operational and widely deployed P2P file-sharing systems.*

Most P2P systems in use today [8, 13] adopt fully distributed and largely unstructured overlays. In such *unstructured* systems there are few constraints on the over-

lay construction and data placement: peers set up overlay connections to a (mostly) arbitrary set of other peers they know, and shared objects can be placed at any node in the system. While the resulting random overlay structures and data distributions may provide high resilience to the degrees of transiency (i.e., churn) found in peer populations, they limit clients to nearly “blind” searches, using either flooding or random walks to cover a large number of peers.

Structured, or DHT (Distributed Hash Table)-based protocols [28, 33, 36, 25], on the other hand, reduce the cost of searches by constraining both the overlay structure and the placement of data – data objects and nodes are assigned unique identifiers or keys, and queries are routed based on the searched object keys to the node responsible for keeping the object (or a pointer to it). Although the resulting overlay provides efficient support for exact-match queries (normally in $O(\log(n))$), this may come at a hefty price in terms of churn resilience, and the systems' ability to exploit node heterogeneity and efficiently support complex queries.

This paper reports on a detailed, measurement-based study of two operational file-sharing systems – the unstructured Gnutella [8] network, and the structured Overnet [23] network. In a closely related effort, Castro et al. [5] presents a simulation-based, detailed comparison of both approaches using traces of Gnutella nodes arrival and departures [30]. Our study complements their work, focusing on the *characterization* – not comparison – of two operational instances of these approaches in terms of resilience, query and control message overhead, query performance, and load balancing.

Some highlights of our measurement results include:

- Both systems are efficient in terms of control traffic (bandwidth) overhead under churn. In particular, Overnet peers have surprisingly small demands on bandwidth.
- While both systems offer good performance for

exact-match queries of popular objects, Overnet surprisingly yields almost twice the success rate of Gnutella (97.4%/53.2%) when querying for a set of shared objects extracted from a Gnutella client.

- Both systems support fast keyword searches. Flooding in Gnutella guarantees fast query replies, especially for highly popular keywords, while Overnet successfully handles keyword searches by leveraging its DHT structure.
- Overnet does an excellent job at balancing search load; even peers responsible for the most popular keywords consume only 1.5x more bandwidth than that of the average peer.

We validate our findings and further extend our conclusions (Sections 7 and 8) through additional measurements as well as detailed analysis and simulation experiments. The measurement and characterization of the two large, operational P2P systems presented in this paper will shed light on the advantages/disadvantages of each overlay approach and provide useful insights for the design and implementation of new overlay systems.

After providing some background on unstructured and structured P2P networks in general and on the Gnutella and Overnet systems in particular, we describe our measurement goals and methodology in Section 3. Sections 4-6 present and analyze our measurement results from both systems. Section 9 discusses related work. We conclude in Section 10.

2 Background

This section gives a brief overview of general unstructured and structured P2P networks and the deployed systems measured in our study – Gnutella and Overnet.

2.1 The Gnutella Protocol

In unstructured peer-to-peer systems, the overlay graph is highly randomized and difficult to characterize. There are no specific requirements for the placement of data objects (or pointers to them), which are spread across arbitrary peers in the network. Given this random placement of objects in the network, such systems use flooding or random walk to ensure a query covers a sufficiently large number of peers. Gnutella [8] is one of the most popular unstructured P2P file-sharing systems. Its overlay maintenance messages include *ping*, *pong* and *bye*, where *pings* are used to discover hosts on the network, *pongs* are replies to pings and contain information about the responding peer and other peers it knows about, and *byes* are optional messages that inform of the upcoming closing of a connection. For query/search, early versions of Gnutella employ a simple *flooding* strategy, where a query is propagated to all neighbors within a cer-

tain number of hops. This maximum number of hops, or *time-to-live*, is intended to limit query-related traffic.

Two generations of the Gnutella protocols have been made public: the “flat” Gnutella V0.4 [7], and the newer loosely-structured Gnutella V0.6 [14]. Gnutella V0.6 attempts to improve query efficiency and reduce control traffic overhead through a two-level hierarchy that distinguishes between superpeers/ultrapeers and leaf-peers. In this version, the core of the network consists of high-capacity superpeers that connect to other superpeers and leaf-peers; the second layer is made of low-capacity (leaf-) peers that perform few, if any, overlay maintenance and query-related tasks.

2.2 The Overnet/Kademlia Protocol

Structured P2P systems, in contrast, introduce much tighter control on overlay structuring, message routing, and object placement. Each peer is assigned a unique hash ID and typically maintains a routing table containing $O(\log(n))$ entries, where n is the total number of peers in the system. Certain requirements (or invariants) must be maintained for each routing table entry at each peer; for example, the location of a data object (or its pointer) is a function of an object’s hash value and a peer’s ID. Such structure enables DHT-based systems to locate an object within a logarithmic number of steps, using $O(\log(n))$ query messages. Overnet [23] is one of the few widely-deployed DHT-based file-sharing systems. Because it is a closed-source protocol, details about Overnet’s implementation are scarce, and few third-party Overnet clients exist. Nevertheless, some of these clients, such as MLDonkey [21], and libraries like KadC [11] provide opportunities for learning about the Overnet protocol.

Overnet relies on Kademlia [20] as its underlying DHT protocol. Similar to other DHTs, Kademlia assigns a 160-bit hash ID to each participating peer, and computes an equal-length hash key for each data object based on the SHA-1 hash of the content. $\langle key, value \rangle$ pairs are placed on peers with IDs close to the key, where “closeness” is determined by the *XOR* of two hash keys; i.e., given two hash identifiers, x , and y , their distance is defined by the bitwise exclusive or (*XOR*) ($d(x, y) = x \oplus y$). In addition, each peer builds a routing table that consists of up to $\log_2(n)$ buckets, with the i th bucket B_i containing IDs of peers that share a i -bit long prefix. In a 4-bit ID space, for instance, peer 0011 stores pointers to peers whose IDs begin with 1, 01, 000, and 0010 for its buckets B_0 , B_1 , B_2 and B_3 , respectively (Fig. 1). Compared to other DHT routing tables, the placement of peer entries in Kademlia buckets is quite flexible. For example, the bucket B_0 for peer 0011 can contain any peers having an ID starting with 1.

Kademlia supports efficient peer lookup for the k clos-

Bucket ID	Common Prefix Length	Cached Peers (Bucket Entries)
B ₀	0	1001, 1100, 1101
B ₁	1	0110, 0100
B ₂	2	0001
B ₃	3	0010

Figure 1: Routing table of peer 0011 in a 4-digit hash space.

est peers for a given hash key. The procedure is performed in an iterative manner, where the peer initiating a lookup chooses the α closest nodes to the target hash key from the appropriate buckets and sends them *FIND_NODE* RPCs. Queried peers reply with peer IDs that are closer to the target key. This process is thus repeated, with the initiator sending *FIND_NODE* RPCs to nodes it has learned about from previous RPCs until it finds the k closest peers. The XOR metric and the routing bucket's implementation guarantee a consistent, $O(\log(n))$ upper bound for the hash key lookup procedure in Kademlia.¹

Overnet builds a file-sharing P2P network with an overlay organization and message routing protocol based on Kademlia. Overnet assigns each peer and object a 128-bit ID based on a MD4 hash. Object search largely follows the *FIND_NODE* procedure described in the previous paragraph with some modifications. We will introduce additional details on Overnet's search mechanism as we present and analyze its query performance in Section 5.

3 Measurement Goals and Methodology

Our study focuses on the *characterization* of two operational instances of the unstructured (Gnutella) and unstructured (Overnet) approaches to P2P networks in terms of churn resilience (Section 4.1), query and control message (Sections 6 and 4.2) overhead, query performance (Section 5.1 and 5.2), and load balancing (Section 6.1). A fair head-to-head comparison of the two deployed systems would be impossible as one cannot control key parameters of the systems such as the number of active peers, the content and the query workload.

We employed a combination of passive and active techniques to carry out our measurement study. For Gnutella, we modified Mutella-0.4.3 [22], an open-source Gnutella client. Mutella is a command-based client, which conforms to Gnutella specifications V0.4 and V0.6. Our measurements of Overnet are based on a modified MLDonkey [21] client, an open-source P2P client written in Objective Caml, that supports multiple P2P systems including Overnet. Modifications to both clients included extra code for parameter adjust-

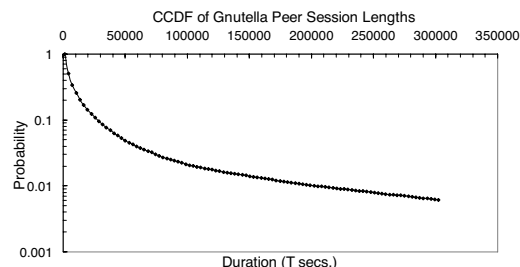


Figure 2: Peers' session length in Gnutella.

ment, probing and accounting, among others. None of these modifications affected the outcome of the collected metrics.

Our modified Gnutella and Overnet clients, each performing a similar batch of experiments, were instantiated at four different locations² around the world and run concurrently to identify potential geographical biases and factor out time-of-day effects from our measurements. All experiments were conducted from April 1st to the 30th, 2005. For brevity, unless otherwise stated, the data presented in the following sections as well as the associated discussions are based on clients placed behind a DSL connection in Evanston, Illinois. Measurements and analysis from the remaining three sites yield similar results and will be briefly discussed in Section 7.

4 Churn and Control Traffic Overhead

The transiency of peer populations (*churn*), and its implications on P2P systems have recently attracted the attention of the research community. A good indication of churn is a peer's session length – the time between when the peer joins a network until it subsequently leaves. Note that a single peer could have multiple sessions during its lifetime by repeatedly joining and leaving the network. We performed measurements of session length for peers in both the Gnutella and Overnet networks, and studied the level of churn of these two systems.

In the context of file-sharing P2P systems, the level of replication, the effectiveness of caches, and the spread and satisfaction rate of queries will all be affected by how dynamic the peers' population is [1, 3, 6, 15, 27]. For P2P networks in general, control traffic overhead is also a function of the level of churn. Control traffic refers to protocol-specific messages sent between peers for overlay maintenance, including peers joining/leaving the overlay, updating routing tables or neighbor sets, and so on. It does not include any user-generated traffic such as query request and replies. In this section, we study the control traffic overhead for both networks and discuss our findings.

4.1 Level of Churn

We performed session-length measurements of Gnutella by modifying the Mutella client [22]. We first collected a

large number of $(IP, port)$ tuples for Gnutella peers by examining ping/pong messages that our Mutella client received. From the set of all the $(IP, port)$ tuples, we probe a randomly selected subset to collect data representative of peers' session-length distribution for the Gnutella network. While performing the measurement, our client periodically (every 20 minutes) tries to initiate a Gnutella-specific connection handshake with each peer in the list. The receiving peer at the probed IP and $port$, if active, either accepts or refuses the connection request (indicating its "BUSY" status). Our session length measurement for Gnutella lasted for 7 days, and captured approximately 600,000 individual peer sessions.

Session-length probing for Overnet was conducted using our modified Overnet (MLDonkey) client [21]. Each peer in Overnet is assigned a unique 128-bit hash ID that remains invariant across sessions. To search for a particular user by hash ID, Overnet provides the *Overnet-Search* message type. Peers connect using a *Overnet-Connect* message; when a peer receives an *OvernetConnect* message, it responds with an *OvernetConnectReply*, which contains 20 other peers' IDs known by the replying peer. To begin the session-length measurement, we collected the hash IDs of 40,000 Overnet peers by sending *OvernetConnect* messages and examining IDs contained in the corresponding reply messages. We then periodically probed these peers to determine whether they were still online. Since it is possible for a peer to use different $(IP, port)$ pairs for different sessions, we rely on *OvernetSearch* messages to iteratively search for and detect peers that start new sessions using different $(IP, port)$ tuples. As in the case of Gnutella, the session length measurement for Overnet also lasted 7 days. During this time we continuously probed 40,000 distinct peers and measured over 200,000 individual sessions.

Figures 2 and 3 give the Complementary Cumulative Distribution Function (CCDF) of peers' session lengths for Gnutella and Overnet, respectively. Figure 2 shows that peers in the Gnutella network show a medium degree of churn: 50% of all peers have a session length smaller than 4,300 seconds, and 80% have session lengths smaller than 13,400 seconds (< 4 hours). Only 2.5% of the session lengths are longer than one day. The median session length of an Overnet peer (Fig. 3) is around 8,100 seconds, 80% of the peers have sessions that last less than 29,700 seconds, and 2.7% of all session lengths last more than a day. Overall, the Overnet network has a measurably lower, but still similar, level of churn when compared to the Gnutella network. In the next section, we analyze the impact of these levels of churn on control traffic overhead for each of these systems.

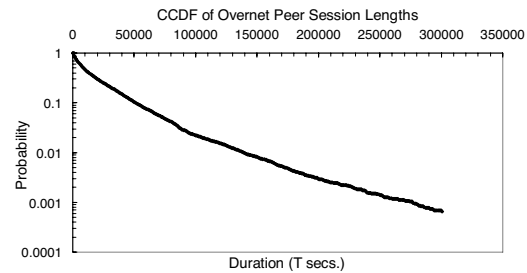


Figure 3: Peers' session length in Overnet.

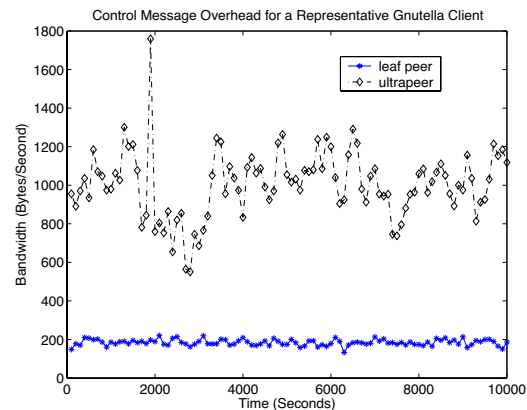


Figure 4: Bandwidth consumption of control messages for the Gnutella client.

4.2 Control Traffic Overhead

In this section, we present and discuss a three-day measurement of the control traffic for Gnutella and Overnet. To better illustrate changes on bandwidth demands at finer time resolution, each figure in this section shows a representative measurement window of 10,000 seconds taken from a client behind a DSL connection in Evanston, Illinois. Each data point corresponds to the average bandwidth demands for every 100 seconds. Data on bandwidth demand for other measurement periods and measurement sites produced similar results.

The measurement of the Gnutella network was done for Gnutella V0.6 using our modified client, which can act either as a leaf peer or as an ultrapeer. The modified client records all control-related messages that it generates in addition to those messages originating from other peers and routed through it. The majority of control-related messages in Gnutella are pings and pongs, along with small percentages of other types of control messages such as those used to repair routing tables. We opted for Gnutella V0.6 as this is the most common version in the Gnutella network.

Gnutella uses ultrapeers to exploit peers' heterogeneity in attributes such as bandwidth capacity and CPU speed, thereby making the system more scalable and efficient. Ultrapeers typically connect to a larger number of neighbors than do leaf peers and they are assigned more responsibility for responding to other peers' messages,

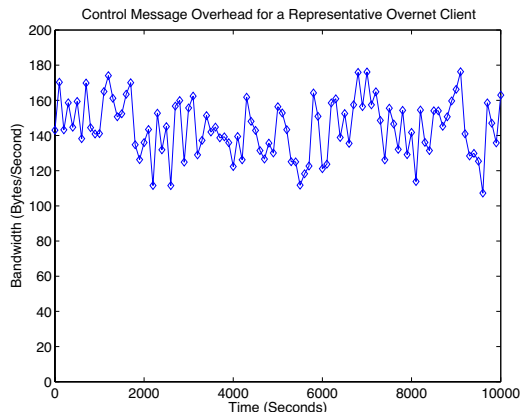


Figure 5: Bandwidth consumption of control messages for the Overnet client.

thus consuming several times more bandwidth for control messages than leaf peers. Figure 4 illustrates this for a leaf peer connected to no more than 4 ultrapeers, and an ultrapeer connected to a maximum of 5 ultrapeer neighbors and 8 leafpeer children. As would be expected, while a leaf peer typically only consumes around 200 Bytes/second for control messages, an ultrapeer normally needs to contribute 5 to 6 times more bandwidth for control traffic (between 800 and 1,400 Bytes/second). Despite this high relative difference between peer types, the bandwidth consumption for an ultrapeer is still reasonably low and never exceeds 2,000 Bytes/second in our measurement.

Overall, a Gnutella peer does not consume a large amount of bandwidth for control-related messages, as it would be expected given the loose organization of its overlay infrastructure. Peers joining and leaving the Gnutella network have little impact on other peers or on the placement of shared data objects, and thus do not result in significant control traffic.

Figure 5 shows the control traffic overhead for our modified Overnet client. Contrary to common belief, we found that Overnet clients consume surprisingly little bandwidth: only around 100 to 180 Bytes/second. The control message overhead for a peer is determined by a number of factors, such as the peer's number of neighbors, the peer's (and its neighbors') probing intervals, the size of the control message, etc. It is thus difficult to directly compare control overhead across different protocols. Nevertheless, the reader should consider that while the measured Gnutella client has a significantly shorter probing interval (10 seconds), it also limits the number of neighbors to 13 (in the ultrapeer case). The Overnet client, on the other hand, often has over 300 neighbor peers in its buckets, which are probed at 1,800-sec intervals.

Although a structured (DHT-based) system has strict rules for neighbor selection and routing table mainte-

nance, these invariants can be maintained in a variety of ways, resulting in quite different levels of control message overhead for different DHT-based systems. Recall that an Overnet peer p only needs to maintain certain numbers of buckets to peer with others and perform routing. Any peer whose hash ID shares the first bit with that of peer p can be in the bucket B_1 of p . Moreover, an Overnet peer never immediately repairs bucket entries corresponding to peers that have left the system. Instead, it fills the missing entries either by periodically asking some of its active neighbors for fresh peers to fill the entries or by performing lazy updates when the peer receives control- or query-related messages for a peer whose key matches an empty bucket entry. Interestingly, this flexibility built into the Overnet's bucket entry system shares much in common with the routing table entry flexibility in Pastry [5], while the periodical, lazy bucket entry repair is quite similar to the periodical, passive routing table (or leaf set) maintenance and repair mechanism that MSPastry [4] and Bamboo [27] employ. Their simulation results and our measurement data on the large, operational Overnet demonstrate the low control message overhead for a DHT-based system with these two properties.

5 Query Performance

Querying is a key component of any P2P file-sharing network, and the effectiveness and the scalability of queries largely determine the user's experience. Queries can be broadly divided into two categories: keyword searching and exact-match queries. Keyword searching is an attempt to find one or more objects containing a specific keyword, while an exact-match query aims at specific objects. Exact-match queries can be further classified as searches for popular (and thus highly replicated) objects and for rare objects (or *needles*) which appear on very few peers. In this section, we evaluate query performance for all three cases in both networks.

Query performance is evaluated in terms of the following three metrics: *Query Success Rate*, the ratio of successful queries to all queries, *Query Resolution Time*, the time between query submission and the arrival of the first query reply, and *Z Query Satisfaction Time* [35], the time it takes for a query to get back at least Z query hits. The last metric is mainly used for keyword search, since it is desirable for such a search to return multiple objects with the requested keyword.

5.1 Keyword Searching

In a P2P file-sharing network, a user often wants to perform a broad search for one or more keywords (such as *Piazzolla*, *Adios Nonino* or *US Open*) to get a list of possible interesting objects before deciding what to download. Intuitively, unstructured P2P systems appear to be

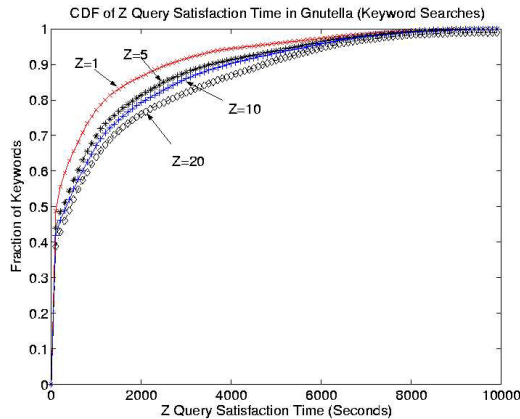


Figure 6: Keyword search in Gnutella.

better suited for keyword searching than do structured ones. For example, a simple implementation of keyword searching uses flooding to propagate the query, and requires that each peer check its local file index for possible matches and reply to the request originator if a match is found. Structured systems, on the other hand, usually have single one-to-one mapping between each individual object and one particular peer that is based on the closeness of the object and the peer hash.

To evaluate the efficiency of each system in supporting keyword searching, we performed experiments using our modified peer clients. For each system, our client sequentially issued queries for 10,000 different keywords. For each search we record the time we start the search and the time when a query hit matching the keyword is received. Since a keyword search can often return multiple query hits indicating different objects containing that keyword, we record all hits for that keyword.

Note that the keywords used were the 10,000 most popular keywords extracted from the query strings routed through our Gnutella client. Thus, our measurement methodology may give some performance advantages to Gnutella, since it is possible that some keywords searched by Gnutella clients never appear in the Overnet system.

Figure 6 shows the CDF of Z query satisfaction time of searches for the 10,000 keywords in Gnutella. Each curve depicts a CDF of query satisfaction times for a particular query satisfaction level (Z). As can be seen, 50% of all keyword searches can be resolved in 100 seconds when $Z = 1$, but it takes over 400 seconds for the same percentage of searches to be satisfactory when $Z = 10$.

Although an unstructured system can easily support keyword searches, it may not be able to do this in the most efficient manner. Recall that for flooding to be scalable, a TTL value must limit the scope of a search. While this may not cause any problems for popular keywords, for less popular ones a single controlled flooding may not be able to find enough peers that collectively contain Z

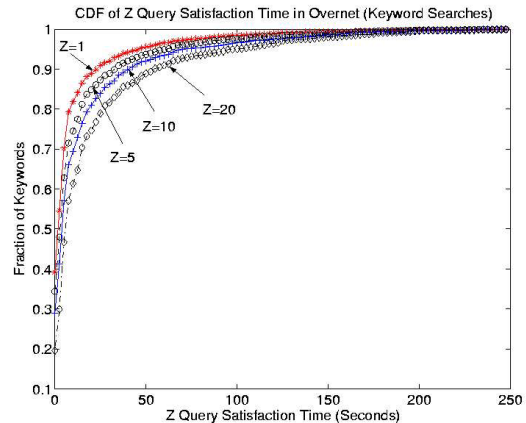


Figure 7: Keyword search in Overnet.

different matching objects.

Figure 7 gives the CDF of Z query satisfaction time for keyword searches in Overnet. Surprisingly, the DHT-based Overnet does an excellent job: 50% of all searches can be resolved in 2.5 seconds with at least a single query hit. Even when $Z = 20$, half of all queries can be satisfied in around 7 seconds.

A careful inspection of the MLDonkey source code and relevant documentation [20, 11] revealed the reason behind these results. To publish keywords for a shared object in Overnet, a client first parses the object name into a series of keyword strings. For each keyword string, the client sends the object's metadata block (containing the object's name, file type and size, MD4 hash of the content, etc) to a certain number of peers with hash IDs close to the keyword's MD4 hash. Note that this technique does not require a "perfect" match between the keyword hash and the peer ID. In Section 6, we will show that this type of inexact match is important for load balancing and search efficiency in Overnet.

When a user performs a search for one of the keywords, the query will be directed to peers whose hash IDs are close enough to it. As a result of the keyword publishing process just described, these peers are very likely to store metadata blocks for objects that match the keyword. For a search that contains multiple keywords the peer can simply filter the results based on the additional keywords. Upon receiving the query, a peer with matching keywords returns a query hit to the initiator and also sends the metadata for the objects matching the keyword(s).

5.2 Exact-match Queries

The performance of exact-match queries is another important factor in determining the user experience. With the results from a keyword search, a user typically chooses one or more specific objects to download; at this point, exact-match queries are performed to locate and download these objects.

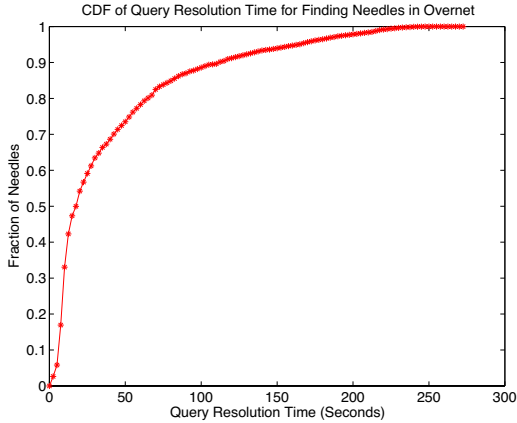


Figure 8: Finding “needles” in Overnet.

Previous work has shown that the object download process often takes from several hours to days or even weeks to complete [29, 9]. During this period, some peers providing a specific shared object may be too busy to handle clients’ download requests, or may become disconnected after some time. To improve download times in this environment, most recent P2P systems support parallel downloading, where a downloading peer can fetch multiple, different pieces of the same object from various peers. Thus, the ability of a peer to continuously search and find alternative sources for a particular object is extremely important for fast and reliable downloads.

5.2.1 Finding Rare Objects – “Needles”

We now look at the performance of Overnet and Gnutella when searching for rare objects. We do this using a worst-case approach, evaluating system performance when each object being searched for has exactly one replica in the entire system.

For each experiment, we ran our modified Gnutella or Overnet clients at the four different measurement locations (Section 3). We alternatively name the four clients A, B, C and D. Peer A shares 1,000 different, randomly generated binary files, each of which has a 50-byte long, random string-based file name; these files are stored only on peer A. Peers B, C and D each issue queries searching for these 1,000 objects. Upon receiving a query hit from peer A, we mark the search as successful and record the time it took to resolve the query. Overall, we are interested in not only the success rate of such queries, but also the resolution speed of successful queries. The process is repeated 10 times before choosing a new node A for the next “round”. In total we ran four rounds of experiments in both Gnutella and Overnet.

Figure 8 shows the query performance for finding “needles” in Overnet. Clearly, our Overnet client does an impressive job at finding “needles” – over half of the needles can be found in 20 seconds, while 80% of them can be located within 65 seconds.

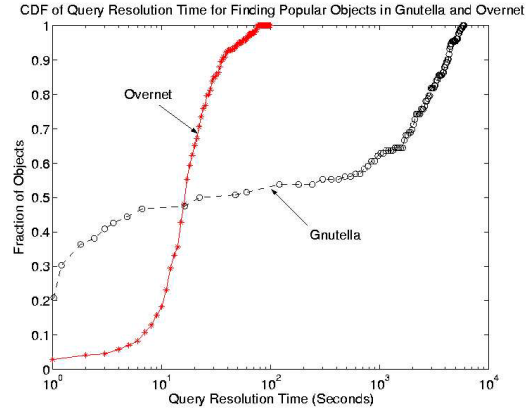


Figure 9: CDF of query resolution time for 20,000 distinct objects from a Gnutella and a Overnet client (x-axis is in log-scale).

Peers whose hash IDs are the closest to that of a shared object will be responsible for storing the object pointer (i.e. the location of the peer sharing the object). Exact-match queries in Overnet first perform the node lookup procedure (Section 2) using the object hash key as the target, then obtain object pointers from these peers to finish the query. Due to the DHT structure, this procedure can be efficiently finished in $O(\log(n))$ steps. One minor difference with other DHT-based systems is that pointers for a specific object not only exist at the peers whose IDs are the closest, but also at peers whose IDs are “close enough” to the object hash. As with the case of keyword publishing and searching (Subsection 5.1), this results in a faster search as well as better load balancing.

The performance for finding “needles” from our Gnutella client is significantly worse than expected: an average success rate of about 1.5% across all four rounds of the experiments. Recall that each search in Gnutella is a “controlled” flooding process; thus, it is possible for such a query never to reach peer A, host to the sole replica of the searched object. The lack of mapping between objects and their locations in unstructured systems makes the task of finding needles particularly difficult, even when using flooding as the query mechanism.

5.2.2 Finding Popular Objects

To study the performance of exact-match queries for popular objects, we examined query hit messages from one of our Gnutella clients and extracted 20,000 distinct shared objects. We then performed queries for all these objects on both the Gnutella and Overnet networks using our modified clients and measured their query performance.

Figure 9 shows the CDF of the query satisfaction time for finding these objects in both systems. We use a log scale for the x axis to accommodate the wide time scale over which queries are resolved. Note that each curve

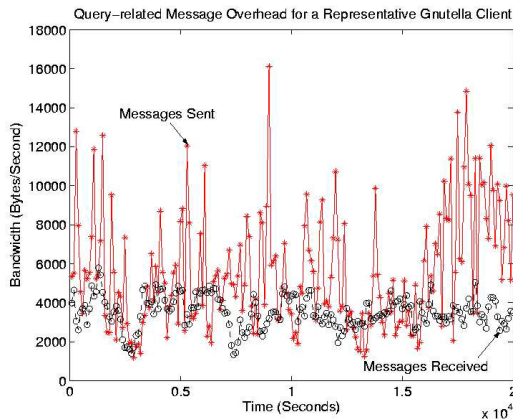


Figure 10: Query-related message overhead for a Gnutella client.

in the figure is normalized by the number of successful queries; failed queries are not considered. Both systems can usually return results for such queries very quickly: 50% of all successful queries are finished within 47 seconds in Gnutella and in less than 17 seconds in Overnet. This is not surprising since Gnutella quickly floods a large number of peers within a few hops, while Overnet takes advantage of its $O(\log(n))$ search mechanism for each object. A particularly interesting difference, however, is the query success rate in each system – while the Overnet client successfully resolves 97.4% of all these queries, the Gnutella client only yields a success ratio of 53.2%, where most failed searches were for objects that are relatively less popular.

6 Query-related Traffic Load

Query-related traffic includes query messages and query replies and can be generated either by the peer itself due to its own queries, or by some other peers for which the peer needs to forward, route, or answer query-related messages. Thus, the query traffic load serves as a good indication of query efficiency and system scalability.

Note that the numbers we present here include only the query or query-reply messages our client received or sent on behalf of other peers. In other words, our Gnutella or Overnet client did *not* issue any queries during the measurement, which removes the potential bias introduced by our own queries. We performed this measurement for both Gnutella and Overnet at all our four measurement sites, with each measurement lasting for about three days. Each figure presented below shows only a representative measurement window of 10,000 seconds taken from our measurement client behind a DSL line in Evanston, Illinois. Each data point corresponds to the average bandwidth for every 100 seconds. Bandwidth data from other measurement periods and other sites are similar.

Figure 10 gives the bandwidth consumption of query-

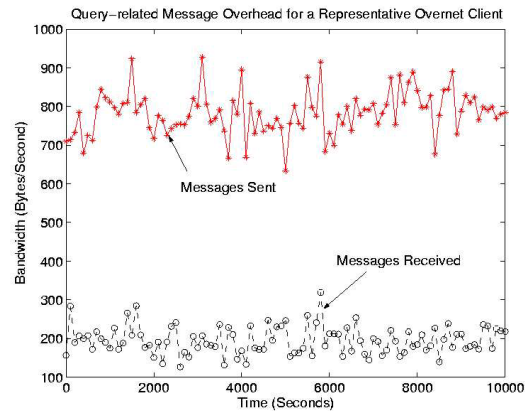


Figure 11: Query-related message overhead for a Overnet client.

related traffic for the Gnutella client. Note that we only give numbers for the case of ultrapeer, since leaf peers in Gnutella do not receive or relay any query messages on behalf of other peers. Similar to Figure 4, our Gnutella peer has up to 5 ultrapeer neighbors and 8 leaf-peer children. As the figure shows, our client typically consumes 5 to 10 KBytes/second for sending query and query-hit messages, and 2 to 6 KBytes/second for receiving such messages. Clearly, even under the default settings of our Gnutella implementation (Mutella) of an ultrapeer, query-related traffic could overwhelm a dial-up a modem user and can only be safely handled by broadband peers. This result is not surprising given the flooding query mechanism at the ultrapeer layer of the network. Overall, Gnutella does a reasonable, but not excellent job for query efficiency and query traffic load. We will see later in Section 8 that the introduction of ultrapeers in unstructured systems greatly improved query success rate and system scalability compared with the purely flat, old Gnutella V0.4, making the inherently unscalable flooding query mechanism acceptable in practice.

Upon receiving a query message in Overnet, a peer will select, from its own buckets, peers whose hash IDs are among the closest to the target keyword or file hash of the search, and send these peers' identities back to the query initiator. Additional query messages could then be sent to those peers during the next iteration of the search until the query is resolved. Figure 11 gives the bandwidth consumption of our Overnet client for sending and receiving all these types of query-related messages. Queries here incur much less overhead than in the case of Gnutella: our Overnet client only consumes around 200 Bytes/second for incoming query messages, and 700 to 900 Bytes/second for sending out replies. Queries in Overnet only involve up to $O(\log(n))$ iterative steps, with the number of messages involved in each step being at a constant level and never growing exponentially as in the case of flooding. As a result, our Over-

net client only needs about one tenth the bandwidth of our Gnutella ultrapeer for handling query-related messages.

Both the Gnutella and Overnet clients consume more outgoing bandwidth than incoming bandwidth for query traffic. Recall that our measurements here are passive. Both our Gnutella ultrapeer in Figure 10 and the Overnet peer in Figure 11 do not initiate any queries. On the other hand, they need to respond to other peers' query messages with replies. Since query reply messages in both systems are typically larger than query messages, more bandwidth thus needs to be consumed for the outbound traffic in our passive measurement.

6.1 Load Balancing

The query load measurement results presented so far are representative of peers with typical configurations in Gnutella and Overnet. Load balancing is critical to system scalability.

Due to the flooding approach to queries, the main contributor of query load for a Gnutella peer is the large number of queries and replies that this peer has to forward. A peer's query load thus largely depends on its number of neighbors. To verify this, we conducted a measurement on our ultrapeer client, where we initially limit the number of its neighbors to 5, then gradually increase the number of neighbors to 100. We record the average query message bandwidth consumption with different number of neighbors. As expected, the query load of the client grows roughly linearly with increasing number of neighbors. In fact, when the number of neighbors exceeds 50, the outgoing bandwidth consumption alone at our ultrapeer consistently rises above 50 KBytes/second, a volume that cannot be handled by common DSL connections. Therefore, query load at different Gnutella peers can be highly skewed and greatly affected by their outdegrees. Still, a peer can easily adjust its query load by changing its number of neighbors.

Previous research reveals that object popularity in P2P systems is highly skewed, and can be well modeled as a Zipf distribution [32]. Thus, for Overnet peers, the main concern about load balancing is that some peers may have IDs that are very close to those of some highly popular keywords, making them potential hot spots of large number of keyword searches. To study this, we change our Overnet client's hash key to be the same as different keywords with different popularities, so that it would be responsible for storing pointers of objects containing these keywords. Due to the diversity of the hash keys for different keywords, we only change the peer's hash key to that of one popular keyword at a time, and test how the query load changes. Overall, we modified our Overnet client to change its own hash ID every 2 hours to be the same as a particular keyword, and mea-

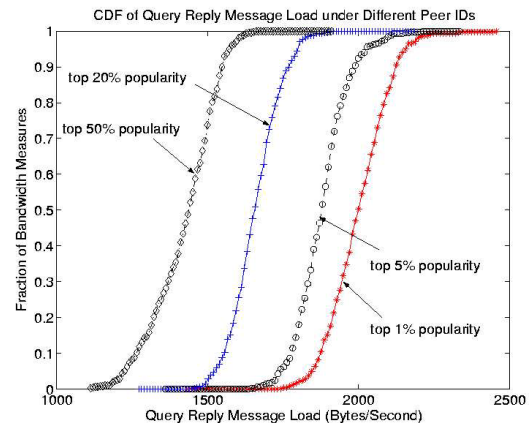


Figure 12: CDF of query-related traffic load when the Overnet client is responsible for different keyword groups with different popularities.

sured its query load during that period. We tested 200 different keywords in four groups, each group containing 50 keywords with similar popularity and different groups representing different popularities. All 50 keywords in Group A belong to the top 1% most popular keywords, while those in Groups B, C, and D belong to the top 5%, 20% and 50% most popular keywords, respectively. These 200 keywords are a small subset of the 10,000 keywords used in Subsection 5.1, and the keyword popularity is based on the local observation of our Gnutella client by examining all query strings that come through.

Figure 12 gives the CDF of query traffic bandwidth consumption for our Overnet client. Each curve corresponds to the CDF of average bandwidth consumption during which our Overnet client has been sequentially assigned the same hash ID as each of the 50 keywords in a particular popularity group. Surprisingly, we did not observe any significant query traffic load for any of these data points: the load for Group A (the top 1% most popular) is generally only 50% higher than that of Group D (50% most popular, or equivalently, median popularity) under different percentages. On the other hand, the highly skewed keyword popularity distribution indicates that the most popular keywords should get at least one order of magnitude more queries than keywords with median popularity.

One possible explanation of the aforementioned disparity between query load and keyword popularity is that Overnet does a good job at distributing query load for a popular keyword among multiple peers whose hash ID is close enough to the hash of the keyword. As briefly mentioned in subsection 5.1, Overnet is not restricted to put file pointers containing a given keyword precisely at the peer whose hash ID is the closest to the keyword's hash. Instead, these pointers can be replicated and distributed among multiple peers whose hash keys are close enough

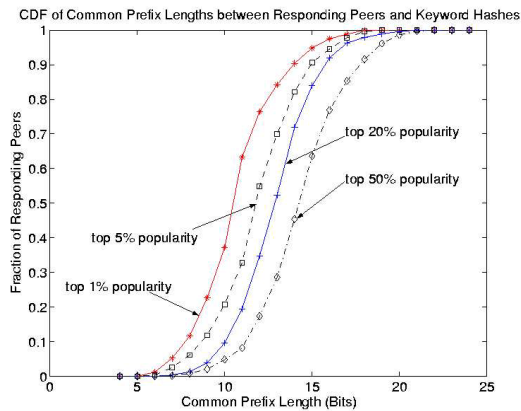


Figure 13: CDF of common prefix lengths between hash keys of responding peers with keyword hashes.

to that of the keyword.

To validate this, we conducted another experiment where we issue searches for keywords in each of the four popularity groups. To each peer that answers a particular keyword search with object pointers, we issue another message to retrieve its hash ID and compare it with the keyword hash to get their common prefix. The longer the common prefix, the closer the two hash keys are.

Figure 13 shows the CDF of the common prefix length between IDs of peers that answered our keyword queries with those of the keywords. Different curves correspond to keyword searches in different popularity groups. Clearly, peers that answer more popular queries tend to share shorter prefixes with the keyword hashes. For example, for keywords in the top-50% popularity range, the common prefix lengths between a keyword and a responding peer is typically between 13 to 15 bits, but often drops to around 10 to 11 bits for the top 1% most popular keywords. This indicates that object pointers for the top 1% most popular keywords are replicated and distributed on about 8 to 16 times more peers than are those of the top 50% most popular keywords. Thus, query load for popular keywords is widely distributed across a larger number of peers in Overnet, effectively achieving load balancing as illustrated in Figure 12.

7 The Impact of Geographical Placement of Peers

Most measurement and discussions presented so far were illustrated using data from our North America measurement site in Evanston, Illinois. To validate that the observed trends and corresponding conclusions are general and do not depend on the particular geographical locations of the measured peers, we repeated our study in all four sites previously mentioned.

For example, Figure 14 depicts the *Z query satisfaction time* of keyword searches using the four measurement sites in three different continents for both Over-

Fraction of Keywords	0.50	0.80	0.90	0.95
Gnutella - U.S.	402.63	2462.57	4511.07	5820.36
Gnutella - Switzerland	421.83	2527.22	4301.53	5574.51
Gnutella - France	430.79	2729.32	4599.89	6220.87
Gnutella - China	483.18	2946.60	4741.86	6184.62
Overnet - U.S.	7.03	27.16	54.97	97.49
Overnet - Switzerland	6.38	23.60	49.12	82.21
Overnet - France	6.65	26.85	51.14	100.47
Overnet - China	7.54	28.19	58.18	109.82

Figure 14: Keyword search performance of four clients across different continents for both the Gnutella and Overnet networks. Numbers shown in the table are *Z* query satisfaction times (in seconds) for different fractions of keywords with $Z = 20$.

net and Gnutella. As the figure shows, at the US measurement site searches for 80% of all keywords can be satisfied in 2,462 and 27.16 seconds, respectively, for Gnutella and Overnet. At the Switzerland site, these two results only change slightly to 2,527 seconds for Gnutella and 23.60 seconds for Overnet. It is clear from the table that search performance across different sites shows only minor difference for either system across the different sites.

In general, all measured data show similarly high degrees of consistency across the different sites. With the same network and same client configurations, for instance, the average control traffic across different measurement sites for both Overnet and Gnutella always remains well below 10%, while the degree of churn of other peers in either Gnutella or Overnet systems as observed by our clients is virtually the same, regardless of the client location.

8 Simulation Study

Having presented measurement data and their analysis for both Gnutella and Overnet, we now use trace-driven simulation of both systems to help further validate and consolidate our findings. Simulations also enable us to obtain aggregated performance numbers of the whole system besides those from individual peers.

Due to space constraints, we only present a small subset of our simulation results. For Gnutella, we explore its message overhead with respect to flooding queries. We also examine the advantage of ultrapeers in terms of system scalability and efficiency. For Overnet, we focus on the lower-than-expected control message overhead and its potential impact on query performance, as well as how effective search and load balancing is attained in Overnet.

All simulations were driven by our collected traces in each system as described in Subsection 4.1. Joins and leaves of peers strictly follow individual sessions captured in the traces. The number of online peers at any time during a simulation ranges between 12,000 and 16,000 for both the Gnutella and Overnet network. Each

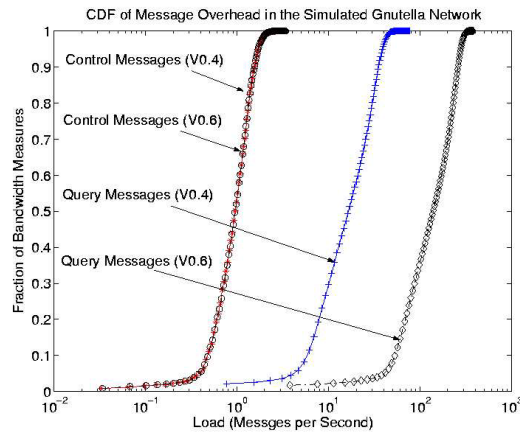


Figure 15: CDF of message overhead for Gnutella V0.4 peers and V0.6 ultrapeers.

simulation runs for one week of simulated time, during which we capture and log various performance numbers including control message overhead, query performance, load balancing condition, etc.

8.1 Gnutella

Our event-based Gnutella simulator was written in 5,000 lines of C++ code. The simulator follows the Gnutella specification, and supports all membership management functionalities as well as query related tasks. We run simulations for both Gnutella V0.4 and V0.6. Each active peer issues a query every 100 seconds on average. Since our network is much smaller than the whole Gnutella network, a smaller value of TTL, 5, is used for flooding. For Gnutella V0.4, each peer is connected with 3 to 10 other peers and has an average outdegree of 5. For Gnutella V0.6, each ultrapeer is connected with 4 leafpeers and 5 other ultrapeers on average.

Figure 15 show the CDF of control message and query message overhead for Gnutella peers in our simulation. Results for both normal peers in Gnutella V0.4 and ultrapeers in Gnutella V0.6 are included. Note that for Gnutella V0.6 we only show numbers for ultrapeers since they dominate the message overhead of leafpeers. As can be seen in the figure, control message overhead for Gnutella is usually very low. Under our simulation settings, it is usually around 1 message/second for both normal V0.4 peers and V0.6 ultrapeers. On the other hand, the flooding query mechanism in Gnutella, although being controlled by the TTL value, results in much higher bandwidth consumption. Normal V0.4 peers typically need to support 5 to 60 messages per second while an ultrapeer could experience 30 to 300 query messages per second. This is not surprising since for Gnutella V0.6, flooding is only restricted to the ultrapeer layer which has a much smaller number of peers than the total population. Thus a flooding query has much higher chance of imposing load on ultrapeers than of reaching normal

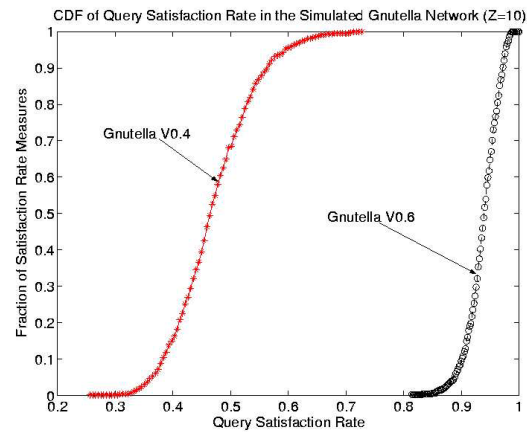


Figure 16: CDF of query success rates for Gnutella V0.4 and V0.6.

peers in V0.4. On the other hand, leaf peers in V0.6 have no query-related responsibility. As a result, the aggregated query loads of all peers for V0.4 and V0.6 stay about the same. Clearly, Gnutella V0.6 takes full advantage of peers' heterogeneity by placing peers with lower bandwidth capacity as leaves.

Another potential benefit of ultrapeers in V0.6 is improved query performance. As already mentioned, flooding in V0.6 is only performed at the ultrapeer layer, which consists of a small fraction of all peers in the system. With similar TTL settings, a query message in V0.6 reaches a much larger fraction of all peers in the network when one considers that each ultrapeer indexes all its leaf peers' shared objects. As a result, query performance in V0.6 is much better than in V0.4. Figure 16 shows the CDFs of keyword *query satisfaction rate* for both Gnutella V0.4 and V0.6, with the same TTL setting of 5. Here we set the value of Z to be 10. We sample query satisfaction rates of all queries in the network every 100 seconds to obtain the CDF curve. Gnutella V0.6 shows a clear advantage, yielding 30-50% higher query satisfaction rates for the same fraction of satisfaction rate measures.

The introduction of ultrapeers not only improves the usability of the system for lower-bandwidth capacity users but also boosts query performance. Nevertheless, the scalability problem of flooding queries in Gnutella V0.4 persists in V0.6.

8.2 Overnet

Our Overnet simulator was written in 3,000 lines of C++ code and supports all membership management, routing, as well as query functionalities of Overnet. We use a 64-bit key space for all hash keys. Each peer maintains a set of buckets storing its neighbor peers, each bucket B_i stores up to 5 neighbors whose hashes share a common prefix length of i with the ID of the host peer. Each peer sends out a query every 100 seconds.

Pinging Interval	200 S	400 S	800 S	1600 S	3200 S
Control Message Overhead	0.5351	0.2669	0.1319	0.0657	0.0328
Valid Bucket Entry Percentage	0.9808	0.8936	0.5679	0.4892	0.3986
Average Query Iterations	2.8826	3.0227	3.4725	3.6021	3.7248
Query Success Rate	1.0000	1.0000	1.0000	1.0000	1.0000

Figure 17: Control Message Overhead and Query Performance for Different Probing Intervals.

To update and populate bucket entries, a peer periodically contacts its neighbor peers, i.e. peers in its buckets, both indicating its own liveness and asking them for identities of other peers. The control message for a peer thus includes these probing messages as well as replies from other peers. The control message overhead, the freshness of bucket peers, and eventually, the efficiency of searches, would all be affected by the frequency of this probing.

Figure 17 illustrates this well. We vary the probing interval from 200 to 3,200 seconds, doubling it at each step. As expected, the average control message overhead of a peer proportionally decreases as one increases the probing interval. As shown in the figure, the reduced control overhead comes at the cost of worse bucket “freshness” and slower query resolution, as indicated by the average number of iterations a query takes. Nevertheless, even when using a probing interval of 3,200 seconds and with a bucket freshness of 0.4, we can still resolve queries very efficiently within a few iterations. This degree of robustness and efficiency can be largely attributed to the construction of bucket peers. Each bucket contains multiple peer entries; even if some peer entries become stale, we can still rely on the remaining fresh entries for the search. Missing or stale peer entries can be easily replaced since prefix matching (Kademlia’s XOR metric) provides enough flexibility for many peers to qualify for a bucket entry. On the other hand, even when using a short probing interval of 200 seconds, the overhead is smaller than 0.5 control messages per second. Control overhead clearly benefits from the bucket entry flexibility and the lazy or periodic repair of buckets in the face of churn. In the following analysis, we employ a probing interval of 800 seconds.

We also explore through simulation how effectively one could balance the load imposed by popular keyword searches. We compare two approaches to object pointer placement, one naive where we simply put pointers to objects on the peer that shares the longest prefix with the keyword, i.e., the “root” peer, and another load-balance conscious one. The load-balancing algorithm applies to any peer x that stores object pointers containing a popular keyword K as follows:

1. Peer x periodically checks its query load for the keyword K . Assume the common prefix length of x and K is L bits.

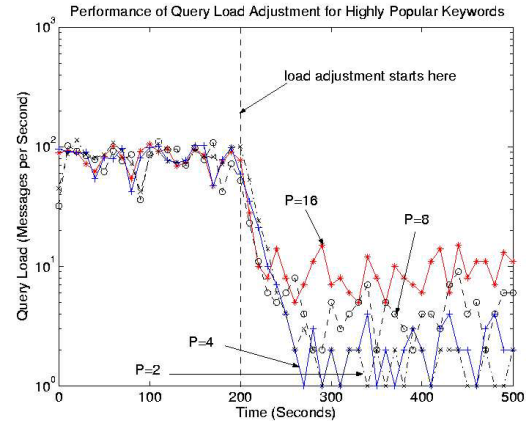


Figure 18: Query load adjustment for the most popular keywords under different load threshold P .

2. If the load is smaller than certain threshold P , go back to step 1. If the load exceeds the threshold, find n other peers whose hashes shared a common prefix of at least M bits with K . The initial value of n is set to 2, and M is set to $L - 1$.
3. Store (some) of the object pointers to these n peers.
4. Set $M = M - 1$, and $n = n * 2$, and start from step 1 again.

For each round of the load-balancing algorithm, we distribute the keyword searching load to more peers by reducing M , the length of the required common prefix between the keyword K and the potential peers that share the load, by 1. Each round thus increases the number of peers that share the query load by a factor of 2, achieving fast and effective load balancing. Even for the most popular keywords, the load balancing algorithm can be finished within a few rounds.

Figure 18 illustrates the responsiveness of load balancing for keyword searches, where we show the average query load for peers that store pointers for a keyword with top 0.1% popularity. We begin by strictly limiting the data pointers on the “root” peer for the keyword. After 200 seconds (shown with a vertical line in the graph), we start the load balancing algorithm by putting object pointers on more qualified peers. The search loads of the keyword under different thresholds P , as indicated by the arrows, can all be reduced to below or around the desired level within 100 seconds.

Load balancing for searches in Overnet is essentially achieved by scattering query load across multiple peers. The more popular a keyword, the more peers will be involved in sharing the load. Figure 19 shows the CDF of common prefix lengths for IDs of peers that answer a keyword search, where each curve corresponds to a set of keywords with certain level of popularity. The load upperbound for keyword search is set to 4 messages per

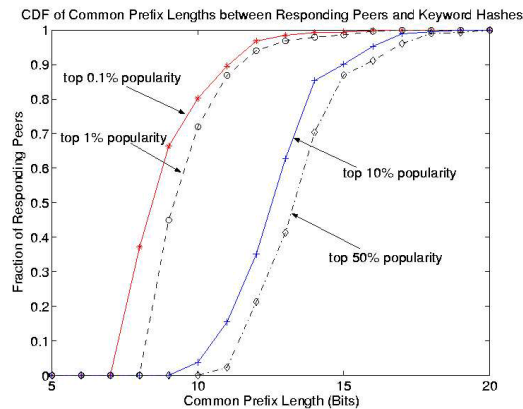


Figure 19: CDF of common prefix lengths between hash keys of responding peers with keyword hashes for keys with different popularities.

second. As expected, the more popular a keyword, the shorter the common prefix length: the common prefix lengths corresponding to the top 0.1% popular keywords are usually 5 to 7 bits shorter than those for the top 50% popular keywords. In other words, we would be able to get query answers from 32 to 128 times more peers for the most popular keywords. Note that our simulation results here are consistent with our measurement-based findings shown in Figure 13.

Load balancing in an Overnet-like system also yields two other desirable consequences. Since object pointers are replicated across multiple peers, there is no need to react to nodes' joins and leaves. These pointers can be lazily replicated to new-born peers to compensate for the pointers lost because of dead peers. In addition, replicated pointers at multiple peers help reduce the average iteration steps that a peer has to take to resolve a query. We have verified this through simulations, but omit our results for brevity.

9 Related Work

The work by Castro et al. [5] compares both unstructured and structured overlay via trace-based simulation, focused on leveraging structural constraints for lower maintenance overhead, exploiting heterogeneity to match different peer capacities and novel techniques of performing floodings or random walks on structured overlays for complex queries. Our study complements their work with the characterization, through extensive measurements, of two operational P2P systems, trying to gain additional insights into the design and implementation of more scalable and efficient peer-to-peer systems.

A lot of measurement works has been done to improve our understanding of different aspects of peer-to-peer file-sharing networks. Saroiu et al. [30] were among the first to perform a detailed measurement study of Napster and Gnutella file-sharing systems using crawlers and probers. In their later works, Gummadi et al. [9] per-

formed a larger-scale P2P workload study for Kazaa at the border routers of the University of Washington, with the main focus on the object content retrieval aspect of the system. Sen et al. [31] also performed aggregated P2P traffic measurement by analyzing flow-level information collected at multiple border routers across a large ISP network, and revealed highly skewed traffic distribution across the network at three different aggregation levels and high dynamics of the three measured P2P systems. Bhagwan et al. [2] present an earlier study of Overnet as they question the usefulness of session times in understanding the availability of objects in a P2P file sharing network. Our study focuses on the measurement-based characterization of two unstructured and structured P2P networks in terms of resilience to churn, control traffic overhead, query performance and scalability. To the best of our knowledge, this is the first study discussing the advantages/disadvantages of each approach based on actual measurement of operational systems.

The more strict routing table and rules in structured P2P systems (i.e., DHTs) compared with unstructured ones motivate the common concern on the potentially large control message overhead and degraded routing performance of structured systems under churn. Bamboo [27] addresses this issue by relying on static resilience to failures, accurate failure detection and congestion-aware recovery mechanisms. Li et al. [16] present a performance versus cost framework (PVC) for evaluating different DHT algorithms in the face of churn. Some other works handling churn include [17, 19]. Our measurement study shows that Overnet does a very good job at handling the level of churn faced by a file-sharing network, yielding low control message overhead and good query performance. It is also interesting to note that Overnet (or Kademlia) already incorporates some important design lessons from previous works on resilient DHTs, including the use of periodic instead of proactive routing structure repair [27], parallel lookups [16], and flexible neighbor selection [5].

Given their key role in the functionality of this class of systems, queries have been actively studied for both the unstructured and structure approaches. A number of proposals have been made to enhance the scalability of queries in Gnutella-like systems. Lv et al. [18] proposes replacing flooding with random walks for queries in such systems. Gia [6] further adopts the idea of topology adaptation, capacity-aware flow control, and biased random walks to both improve query performance and scalability. Our previous work [3, 24] presents additional measurements of Gnutella's peer session lengths and suggests organizational protocols and query-related strategies that take advantage of their distribution to increase system scalability and boost query performance. The reported measurement of Gnutella shows that flood-

ing does indeed result in high overhead, especially for high degree ultrapeers. Applying ideas such as those proposed in [6] and [24] could help improve system scalability.

Some recent work has focused on keyword searches in structured systems [26, 10, 34]. Most of the proposed solutions include the hashing of a keyword to a peer with the closest hash, and the storing of pointers to all objects containing the keyword at this peer. Perhaps one of the biggest concerns with this approach is the load on the peer that stores the pointers, especially if this peer is responsible for very popular keywords. Overnet uses a similar idea to support keyword search, except that instead of mapping a keyword to one particular peer, it replicates object pointers around a collection of peers whose hash IDs are close enough to that of the keyword. As we have shown in Section 6, this implicit load-balancing mechanism turns out to be very effective in distributing the load of popular keyword searches across multiple peers. As a result, keyword searches in Overnet are lightweight, effective, and achieve a balanced load.

10 Conclusion

We presented a multiple-site, measurement-based study of two operational and widely-deployed file-sharing systems. The two protocols were evaluated in terms of resilience, message overhead, and query performance. In general, our findings show that the structured overlay approach, as represented by Overnet, offers good scalability and efficiency in supporting file sharing networks. For example, while both Gnutella and Overnet show relatively low control message overhead, Overnet's overhead is particularly low in part as a result of its flexible bucket management policies. Also, both approaches seem to support fast keyword searches with Overnet successfully guaranteeing efficient, non-trivial keyword searching by leveraging the underlying DHT infrastructure and its $O(\log(n))$ lookup mechanism. Finally, while Gnutella's ultrapeers can adjust their load by changing their outdegree, Overnet's peers implicitly distribute and replicate popular object pointers across multiple peers, achieving excellent load balancing and faster query replies.

We trust that the measurement and characterization presented in this paper will help the community to better understand the advantages and disadvantages of each approach as well as provide useful insights into the general design and implementation of new peer-to-peer systems.

Acknowledgments

We would like to thank Jeanine Casler, and David Choffnes for their invaluable help. In addition, we thank our shepherd Stefan Sariou and the anonymous reviewers who provided us with excellent feedback.

References

- [1] BAWA, M., DESHPANDE, H., AND GARCIA-MOLINA, H. Transience of peers and streaming media. In *Proc. of the 1st Workshop on Hot Topics in Networks (HotNets)* (2002).
- [2] BHAGWAN, R., SAVAGE, S., AND VOELKER, G. M. Understanding availability. In *Proc. of 2nd International Workshop on Peer-to-Peer Systems (IPTPS)* (2003).
- [3] BUSTAMANTE, F. E., AND QIAO, Y. Friendships that last: Peer lifespan and its role in P2P protocols. In *Proc. of 8th International Web Content Caching and Distribution Workshop (WCW)* (2003).
- [4] CASTRO, M., COSTA, M., AND ROWSTRON, A. Performance and dependability of structured peer-to-peer overlays. In *Proc. of the International Conference on Dependable Systems and Networks (DSN)* (2004).
- [5] CASTRO, M., COSTA, M., AND ROWSTRON, A. Debunking some myths about structured and unstructured overlays. In *Proc. of the 2nd Symposium on Networked Sys. Design and Impl. (NSDI)* (2005).
- [6] CHAWATHE, Y., RATNASAMY, S., BRESLAU, L., AND SHENKER, S. Making Gnutella-like P2P systems scalable. In *Proc. of SIGCOMM* (2003).
- [7] CLIP2. The Gnutella protocol specification v0.4. RFC, The Gnutella RFC, 2000.
- [8] GNUTELLA. Gnutella: Distributed information sharing. <http://gnutella.wego.com>, 2003.
- [9] GUMMADI, K. P., DUNN, R. J., SAROIU, S., GRIBBLE, S. D., LEVY, H. M., AND ZAHORJAN, J. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. of 19th Symposium on Operating Systems Principles* (2003).
- [10] HARREN, M., HELLERSTEIN, J. M., HUEBSCH, R., LOO, B. T., SHENKER, S., AND STOICA, I. Complex queries in dth-based peer-to-peer networks. In *Proc. of 1st International Workshop on Peer-to-Peer Systems (IPTPS)* (2002).
- [11] KADC HOMEPAGE. KadC. <http://kadc.sourceforge.net>, 2005.
- [12] KARAGIANNIS, T., BROID, A., FALOUTSOS, M., AND CLAFFY, K. Transport layer identification of P2P traffic. In *Proc. of ACM SIGCOMM Internet Measurement Conference* (2004).
- [13] KAZAA. <http://www.kazaa.com>. 2001.
- [14] KLINGBERG, T., AND MANFREDI, R. Gnutella 0.6. RFC, The Gnutella RFC, 2002.
- [15] LI, J., STRIBLING, J., MORRIS, R., GIL, T., AND KAASHOEK, F. Routing tradeoffs in peer-to-peer DHT systems with churn. In *Proc. of 3rd International Workshop on Peer-to-Peer Systems (IPTPS)* (2004).
- [16] LI, J., STRIBLING, J., MORRIS, R., KAASHOEK, M. F., AND GIL, T. M. A performance vs. cost framework for evaluating dht design tradeoffs under churn. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)* (2005).
- [17] LIBEN-NOWELL, D., BALAKRISHNAN, H., AND KARGER, D. Analysis of the evolution of peer-to-peer systems. In *Proc. of the Symposium on Principles of Distributed Computing (PODC)* (2002).
- [18] LV, Q., CAO, P., COHEN, E., LI, K., AND SHENKER, S. Search and replication in unstructured peer-to-peer networks. In *Proc. of ICS* (2002).
- [19] MAHAJAN, R., CASTRO, M., AND ROWSTRON, A. Controlling the cost of reliability in peer-to-peer overlays. In *Proc. of 2nd International Workshop on Peer-to-Peer Systems (IPTPS)* (2003).
- [20] MAYMOUNKOV, P., AND MAZIERES, D. Kademia: A peer-to-peer information system based on the xor metric. In *Proc. of 1st International Workshop on Peer-to-Peer Systems (IPTPS)* (2002).

- [21] MLDONKEY. MLDonkey Homepage. <http://mldonkey.org>, 2005.
- [22] MUTELLA. <http://mutella.sourceforge.net>. 2003.
- [23] OVERNET. <http://www.overnet.com>. 2003.
- [24] QIAO, Y., AND BUSTAMANTE, F. E. Elders know best - handling churn in less structured P2P systems. In *Proc. of the IEEE International Conference on Peer-to-Peer Computing (P2P)* (September 2005).
- [25] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. of SIGCOMM* (2001).
- [26] REYNOLDS, P., AND VAHDAT, A. Efficient peer-to-peer keyword searching. In *Proc. of IFIP/ACM Middleware* (2003).
- [27] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a DHT. In *Proc. of USENIX Technical Conference* (June 2004).
- [28] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware* (2001).
- [29] SAROIU, S., GUMMADI, K. P., DUNN, R. J., GRIBBLE, S. D., AND LEVY, H. M. An analysis of Internet content delivery systems.
- [30] SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. D. A measurement study of peer-to-peer file sharing systems. In *Proc. of Annual Multimedia Computing and Networking (MMCN)* (2002).
- [31] SEN, S., AND WANG, J. Analyzing peer-to-peer traffic across large networks. In *Proc. of ACM SIGCOMM Internet Measurement Conference* (2002).
- [32] SRIPANIDKULCHAI, K. The popularity of Gnutella queries and its implications on scalability. In *O'Reilly's OpenP2P* (2001).
- [33] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM* (2001).
- [34] TANG, C., XU, Z., AND DWARKADAS, S. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. of SIGCOMM* (2003).
- [35] YANG, B., AND GARCIA-MOLINA, H. Efficient search in peer-to-peer networks. In *Proc. of the International Conference on Distributed Computing Systems (ICDCS)* (2002).
- [36] ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Report UCV/CSD-01-1141, Computer Science Division, UC, Berkeley, 2001.

Notes

¹The Kademlia protocol is discussed in detail in [20].

²Evanston, USA; Zurich, Switzerland; Paris, France and Beijing, China.

Reclaiming Network-wide Visibility Using Ubiquitous Endsystem Monitors

Evan Cooke
University of Michigan
`emcooke@umich.edu`

Richard Mortier, Austin Donnelly, Paul Barham, Rebecca Isaacs
Microsoft Research, Cambridge
`{mort,austind,pbar,risaacs}@microsoft.com`

Abstract

Network-centric tools like NetFlow and security systems like IDSes provide essential data about the availability, reliability, and security of network devices and applications. However, the increased use of encryption and tunnelling has reduced the visibility of monitoring applications into packet headers and payloads (e.g. 93% of traffic on our enterprise network is IPSec encapsulated). The result is the inability to collect the required information using network-only measurements. To regain the lost visibility we propose that measurement systems *must* themselves apply the end-to-end principle: only endsystems can correctly attach semantics to traffic they send and receive. We present such an end-to-end monitoring platform that ubiquitously records per-flow data and then we show that this approach is feasible and practical using data from our enterprise network.

1 Introduction

Network enabled applications are critical to the running of large organisation. This places great importance on monitoring methods that provide visibility into the network. Tools such as NetFlow [4] are becoming essential for efficient network management, and continuous network monitoring platforms are the subject of ongoing research [8, 12]. However, these tools assume that in-network tools have direct access to packet headers and payloads. Unfortunately, the composition of network traffic is changing in ways that directly impact our ability to solve security and network availability problems.

Two trends are particularly troubling for monitoring approaches relying on a network-centric perspective. First, the use of encryption and tunnelling preclude any inspection of payloads; use of tunnelling may also prevent connection information from being determined. Second, the complexity of network applications means that tracking network application behaviour can require visibility into traffic to many destinations using proto-

cols and ports that are expensive to differentiate from other traffic [9]. This is particularly notable in modern enterprise networks where many services are provided over dynamically allocated ports. As a result, collecting all the packets at an upstream router is often no longer sufficient to report on the performance or diagnose problems of downstream network applications. Essentially, network operators have contradictory requirements: security ‘in depth’ through mechanisms leading to opaque traffic (e.g. IPSec) *and* fine-grained auditing only available through traffic inspection.

We propose to provide this fine-grained auditing capability without restricting the ability to deploy essential security mechanisms by using information collected on endsystems to reconstruct an ‘end-to-end’ view of the network. Each endsystem in a network runs a small daemon that uses spare disk capacity to log network activity. Each desktop, laptop and server stores summaries of all network traffic it sends or receives. A network operator or management application can query some or all endsystems, asking questions about the availability, reachability, and performance of network resources and servers throughout the organization. We initially target deployment in government or enterprise networks since these exercise a high degree of control over endsystems. This makes it feasible to deploy a standard operating system image supporting the monitoring facility, and to control data logging in a manner consistent with network security and privacy policies.

Ubiquitous network monitoring using endsystems is fundamentally different from other edge-based monitoring: the goal is to passively record summaries of *every* flow on the network rather than to collect availability and performance statistics or actively probe the network. Projects such as DIMES [5] and Neti@Home[14] use endsystem agents to monitor network properties (e.g. availability and reachability). The Anemone system also collects endsystem data but combines it with routing data to construct a view of the network [13].

In contrast, ubiquitous endsystem network monitoring is more closely related to in-network monitoring approaches like NetFlow [4] in that it provides summaries of all traffic on a network. It also provides a far more detailed view of traffic because endsystems can associate network activity with host context such as the application and user that sent a packet. This approach restores much of the lost visibility and enables new applications such as network auditing, better data centre management, capacity planning, network forensics, and anomaly detection. Using real data from an enterprise network we present preliminary results showing that instrumenting, collecting, and querying data from endsystems in a large network is both feasible and practical.

2 Muddy Waters

For many older network applications it is possible to reconstruct an entire application's session simply by observing the packets between a client and a server (e.g. telnet). The problem is that we have lost much of this visibility due to new network application requirements and deployment practices. Some of this loss can be attributed to new security and privacy features that have rendered a significant amount of network traffic opaque. Furthermore, increasingly complex application communication behaviour can make attributing traffic to specific applications complex and impractical.

2.1 Opaque Traffic

As the Internet has grown to support critical transactions like the transfer of money, security and privacy requirements have come to the forefront of application development and deployment. For example, many large organisations support remote offices and home workers using encrypted VPNs to tunnel traffic back to a central location. Devices that want to monitor this traffic must be placed before or after these tunnels or have access to any keys required to decrypt the traffic.

Two situations arise as a result of this need to monitor network usage. If an organization outsources the running of their network, then they must turn over any keys to their service provider, trusting that the provider will not disclose highly sensitive information. Alternatively, if they keep their network in-house, they must increase its cost and complexity by setting up tens or hundreds of monitoring points as well as the infrastructure required to collect and process the resulting data. Analysing a snapshot of the configuration files from 534 routers in a large enterprise network, we found 193 separate sites that would require integration into such a system – a substantial investment.

Problems with opaque traffic also exist within each office site network. To increase security within their LANs, organisations may use mechanisms such as IPSec to provide authentication of application packets destined for internal addresses. This claim is supported by analysis of an 8-day packet trace collected at a remote office site in large enterprise network. The trace included $\simeq 14,000$ host source addresses, $\simeq 600$ of which were on-site. Over 93% of the collected packets were transport-mode IPSec, and so encapsulated in an ESP header and trailer. This observation differs significantly from previously reported traffic measurements in ISP and educational networks [9, 15], and highlights one of the significant differences between controlled environments like enterprise networks and more open networks: even monitoring devices sitting inside a local LAN may have very limited visibility into network traffic.

2.2 Complex Application Behaviour

Even when a packet is not opaque, it may still be impractical to extract application information and behaviour. Compatibility and security requirements often result in applications that tunnel traffic using common protocols such as HTTP and other transports over varying ports [2, 3].

Modern applications may also exhibit highly complex communication relationships. For example, even an apparently simple application such as email often no longer operates straightforwardly between a given client and server over a single protocol such as SMTP. Instead, “checking your mail” can require connections to many servers such as to an authentication server in order to obtain appropriate credentials, to a mail server to authenticate the user to a mailbox, and finally many other connections to download different headers, mails, and attachments. In addition, the mail application may concurrently be performing background tasks such as synchronising address books, and maintaining calendar alerts. In all, “checking your mail” can instantiate tens of connections to several servers, making it problematic to attribute the relevant traffic to a single application.

In summary, increasingly opaque network traffic and complex application behaviour introduce significant visibility problems for network-centric monitoring approaches. The question is how we can obtain insight into the network in the face of these visibility problems.

3 End-to-End Network Monitoring

To provide the necessary visibility into network traffic we propose an endsystem-based network management platform that uses information collected at the edge to construct a view of the network. Each endsystem in a

network runs a small daemon that uses spare disk capacity to log network activity. Each desktop, laptop and server stores summaries of all network traffic it sends or receives. A network operator or management application can query some or all endsystems, asking questions about the availability, reachability, and performance of network resources throughout an organization.

To validate an endsystem-based monitoring approach we constructed a prototype designed to be integrated into the standard operating system distribution for a large enterprise. We used the Windows built-in kernel monitoring ETW [10] facility to report socket creation/destruction and data transmission/reception. A user-space Windows service processed the ETW events periodically outputting summaries of network data to disk. We mapped each network invocation to a specific application on the endsystem using process identifier information in the common ETW header. Flow tuples were recorded according to the following schema: (*sip*, *dip*, *dport*, *sport*, *proto*, *process_name*, *PID*, *bytes*, *packets*, *timestamp*). This enabled the system to find all the network data associated with complex applications such as Microsoft Outlook and to observe network packets before being encrypted in the VPN layer. A preliminary evaluation of the prototype is described in Section 5.

4 Novel Uses

The fresh perspective provided by an end-to-end monitoring platform enables a range of new network management applications. In contrast to existing monitoring techniques, the platform provides fine-grained usage information in multiple dimensions (host, user, application, virtual machine) as well as the capacity to store this data for significant periods of time.

Network auditing. An operator wishing to determine who is using an expensive WAN link can query the system to determine all the hosts, applications, and users that have used the link over the past few weeks. The system could even be used in a feedback loop to throttle specific applications and hosts using more than their allocated share of a given network resource.

Data centre management. Fine-grained information on network usage can be attributed to individual applications, machines and even specific virtual machines. For example, one might use such a system to account network usage to individual users on a server hosting multiple virtual machines multiplexing a single IP address.

Capacity planning. By using historical data on network usage by specific applications stored on many endsystems, detailed models of application network usage can be built. These can predict the impact of service changes such as distributing email servers among many sites or concentrating them in a few datacentres.

Anomaly detection. Distribution of historical network data across many endsystems also enables new applications that require detailed historical context. For example, models of normal behaviour can be constructed from this extensive distributed archive and deviations from past behaviour detected. An operator could ask for a detailed usage report of all abnormal applications across an enterprise.

5 Feasibility Study

Having argued the need for, and utility of, a monitoring system with better visibility, we now consider several key implementation and deployment issues: (i) where might an end-to-end monitoring system be deployed? (ii) how many endsystems must be instrumented? (iii) what data should be collected? (iv) how can that data be accessed? (v) what is the performance impact on participating endsystems? and (vi) what are the security implications?

(i) System Deployment. One major challenge for an endsystem monitoring platform is how to instrument edge devices. In many large networks, access to network-connected elements is strictly controlled by a central organization. In particular, enterprise and government networks typically have infrastructure groups that generate and enforce policies that govern what machines can be connected to what networks and what software they must run. These highly controlled settings are the perfect environment for end-system monitoring because they are also often quite large and require tools that can provide visibility across a whole network. For example, our own enterprise network contains approximately 300,000 endsystems and 2,500 routers. While it is possible to construct an endsystem monitor in an academic or ISP network there are significant additional deployment challenges that must be addressed. Thus, we focus on deployment in enterprise and government networks that have control over software and a critical need for better network visibility. We discuss the security and privacy implications of collecting endsystem data later.

(ii) Deployment Coverage. Even under ideal circumstances there will inevitably be endsystems that simply cannot easily be instrumented, such as printers and other hardware running embedded software. Thus, a key factor in the success of this approach is obtaining good visibility without requiring instrumentation of all endsystems in a network. Even if complete instrumentation were possible, deployment becomes significantly more likely where incremental benefit can be observed.

If traffic were uniformly distributed between N endsystems, then each additional instrumented endsystem contributes another $1/N$ th to the global view. This observation is initially discouraging as it suggests that a

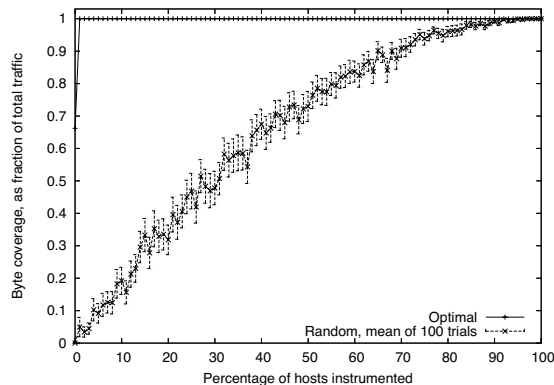


Figure 1: Fraction of traffic observed by endsystem monitoring as increasing subsets of endsystems are instrumented. Error bars on the *random* line show the 95% confidence intervals of the mean.

very high proportion of endsystems would have to participate to obtain a high percentage of traffic. However, if the traffic distribution is less symmetric, a smaller proportion of participating systems may still lead to a useful proportion of the traffic being observed.

To investigate the contribution of different endsystems to the overall traffic we analysed the 8-day network trace from a large enterprise network described in Section 2. We computed the number of bytes and flows observed by each endsystem for the entire trace and for different applications. We define the *byte coverage* as the proportion of the total number of bytes observed by the system across the network within a given time period, and the *flow coverage* similarly for network flows.

Figure 1 depicts the byte coverage as a function of the number of endsystems instrumented. The *optimal* line shows the coverage when endsystems were chosen based on their contribution to the total, largest first. This line shows that instrumenting just 1% of endsystems was enough to monitor 99.999% bytes on the network. This 1% is dominated by servers of various types (e.g. backup, file, email, proxies), common in such networks. Note that in these controlled networks, traffic to and from external addresses will typically traverse some kind of proxy device. With simple modifications to track the setup and teardown of connections for the proxied traffic, these devices can provide excellent visibility into traffic where one side of the connection is not itself part of the controlled network.

The *random* line in Figure 1 shows the mean and 95% confidence intervals across 100 trials when endsystems are selected at random from the total host population. In this case, the resulting coverage is slightly better than linear since both transmit and receive directions are monitored, but variance is quite high due to the fact that if a given trial includes just one of the top 1% endsystems, it leads to a disproportionate improvement in coverage. We also measured flow coverage since that is of interest

for a number of security applications. The results were very similar to the byte coverage.

Finally, since per-application information is a significant benefit of our approach we also analyzed byte and flow coverage for the top 10 applications (for bytes transmitted and received). Again, the results were almost identical to Figure 1: because many enterprise applications are heavily client-server based, it is possible to achieve excellent visibility into them all with just a few instrumented machines. Even if enterprise network workloads become peer-to-peer dominated in the future, a significant shift away from centralized communication models, the worst case is that each extra machine instrumented brings only a $1/N$ improvement (for N the number of end-systems on the network and assuming a completely uniform traffic distribution). Since many centralized applications like security proxies are not suited to peer-to-peer topologies, it is likely that a balance of different topologies will likely persist.

(iii) Data Collection. One common method of storing network data is to capture all packets using a packet sniffer, but this can result in unmanageably large datasets. For example, even a moderately busy server transmitting at 100 Mbps would result in recording gigabytes of data per hour if just the IP and TCP headers were recorded. Since the 1% of endsystems that provide the best coverage are often precisely the busiest 1% of endsystems, a more scalable approach is highly desirable.

The problem of collecting and storing data is well-known in network-centric monitoring. It is often infeasible for routers and other network devices to capture all the packets that they forward so they typically aggregate data, storing information about each *flow* rather than each packet (e.g. NetFlow [4]). Flow records provide excellent compression since a connection with hundreds of packets is synthesised into a single flow. The information in such a flow record might include timestamps, protocol, source, destination, number of packets in the flow, and other fields traditionally available through packet inspection such as TCP headers. Thus, using flows rather than packets provides nice tradeoff between resource cost and network information.

Flows can also be augmented with endsystem information. For example, the user executing the application, the current round-trip time estimates from the TCP stack. Furthermore, the monitoring software can be placed before encryption and tunneling layers so that the resulting flow records store both unobfuscated network activity with host contextual information. Application-level encryption such as SSL may require additional instrumentation of system libraries or applications.

(iv) Accessing Distributed Data Stores. Assuming that we can instrument and collect flows on 300,000 different endsystems, the question becomes how to access

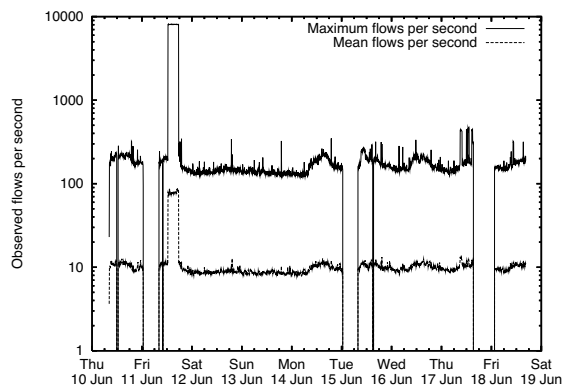


Figure 2: Maximum and mean observed flows per second. The three dropout periods occurred due to crashes of our monitoring system.

these widely distributed data stores in a timely manner. One approach is to collect and centrally store all flow records from all endsystems. This approach clearly doesn't scale well when more than a handful of busy endsystems are involved. Another possibility is to transfer summaries of flows to a central location. The problem is that because the total amount of data is so large, deciding what to summarise requires network applications and operators to already know the questions they wish to ask. This is not always possible. For example, when examining the artifacts left by a network intruder.

A more flexible approach is to provide a platform allowing network management applications to insert queries and receive aggregated responses in real-time. Fortunately, a variety of distributed databases supporting access to network management data already exist [7, 16]. These systems have addressed scalability, the maintenance of the ACID property, support for active database/trigger mechanisms, and the temporal nature of network data. By making each data collection node a member of a large distributed query system one could access data from across the entire system in a timely manner. For example, one could utilize the ability of SDIMS [16] to build dynamic aggregation trees or use hierarchical DHT rings [11] to multicast queries to specific sub-networks.

(v) Endsystem Performance. The busiest 1% of endsystems are often the most useful to instrument so an important concern is the impact of collecting and storing flows on an endsystem. In general, three key resources can be used: memory, CPU, and disk. Memory cost is a function of the number of concurrently active flows, CPU cost a function of the number of packets observed, and disk cost a function of the number of records stored. We now estimate these costs using the packet trace described in Section 2.

Memory Cost. The memory cost on the host is dominated by state required to store flow records before they are exported to disk. To evaluate this cost we constructed

Export Timer (s)	Per-endsystem write rate	
	Mean (kB/s)	Maximum (kB/s)
1	1.445	2081.00
5	0.310	418.00
10	0.168	211.00
30	0.073	71.70
60	0.049	37.50
300	0.022	12.10
900	0.016	5.40
1800	0.016	5.40

Table 1: Per-endsystem maximum and mean rates at which records must be written for a variety of export periods.

a flow database for each of the endsystems. We denote a flow as a collection of packets with identical IP 5-tuple and no inter-packet gap of greater than 90 seconds. Figure 2 depicts the observed maximum and mean flows per second over all endsystems in the trace. Allowing a generous 256 bytes per in-memory flow record, the exceptional worst case memory consumption is under 2 MB, with an average of ≈ 25 kB.

CPU Cost. To evaluate the per-endsystem CPU overhead we constructed a prototype flow capture system using the ETW event system [10]. ETW is a low overhead event posting infrastructure built into the Windows OS, and so a straightforward usage where an event is posted per-packet introduces overhead proportional to the number of packets per second processed by an endsystem. We computed observed packets per second over all hosts, and the peak was approximately 18,000 packets per second and the mean just 35 packets per second. At this rate of events, published figures for ETW [1] suggest an overhead of no more than a few percent on a reasonably provisioned server.

Disk Cost. Finally, we consider disk cost by examining the number of flow records written to disk. Using the number of unique flows observed in a given export period as an estimate of the number of records that would need to be written, Table 1 shows the disk bandwidth required. For example, for a 1 second export period there are periods of high traffic volume requiring a large number of records be written out. However, if the export timer is set at 300 seconds, the worst case disk bandwidth required is ≈ 4.5 MB in 300 seconds, an average rate of 12 kBps. The maximum storage required by a single machine for an entire week of records is ≈ 1.5 GB, and the average storage just ≈ 64 kB. Given the capacity and cost of modern hard disks, these results indicate very low resource overhead.

Initial results from our prototype system are very promising. Costs are within acceptable limits for the handful of key systems required for excellent visibility, and well within limits for most users' endsystems. Moreover, the costs described are not fixed performance requirements and operators could be given the ability to adjust the level of resource usage. For example, one could

add an additional sampling parameter to reduce the number of flows processed and stored or use adaptive flow sampling approaches to further reduce load [6].

6 Security/Privacy Implications

Data available through such a system could potentially support a number of interesting network security and forensics applications. However, several security and privacy questions arise when collecting sensitive network traffic on many endsystems. Specific requirements will depend heavily on the deployment environment, e.g. in a corporate environment there are often regulations governing the protection of data. We now highlight a few of the important issues and suggest how a basic cross-validation strategy can help handle them.

To maintain endsystem integrity and communication security, simple procedures such as privilege separation for software, and encryption of queries may be used. Queries should be authenticated by requiring that they be properly signed by a designated authority, ensuring malicious parties cannot easily discover information about the network to their benefit, and helping prevent malicious or naive users from instigating denial-of-service attacks by introducing many excessively complex or long-running queries. Nefarious insertion or removal of flow data can be detected by asking a host to report on a known quantity, and then validating with the other ends of the flows. For example, a suspicious host could be queried for the amount of data it transmitted and to whom, and the receivers on the same network queried to validate that they received the data that was sent.

Endsystem monitoring also provides additional privacy protection compared to other monitoring approaches. Since each endsystem logs only the data it sends or receives, a node never has access to data that it hasn't already observed. Furthermore, each organisation can customize the data that is logged based on specific endpoints and applications. This is a significant advantage over in-network monitoring solutions where it is difficult to apply privacy filters when the data is recorded requiring that data be scrubbed later. And, an endsystem monitoring solution enables selected highly trusted systems to have different privacy policies than other parts of the network.

7 Conclusion

We believe that network centric monitoring approaches will continue to lose visibility into the network as traffic becomes more opaque and complex. Rather than directly instrument the network, we propose an end-to-end monitoring platform that uses data collected on endsystems to construct a view of the network. Endsystems

are able to provide significantly more visibility than network devices which lack critical host context. An end-to-end platform also enables many new applications like auditing of network resources, better data centre management, capacity planning, network forensics, and anomaly detection. Our preliminary results using real data from an enterprise network show that collecting and querying data from endsystems in a large network is both feasible and practical.

References

- [1] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *6th Symposium on Operating Systems Design and Implementation (OSDI'04)*, Dec. 2004.
- [2] S. Baset and H. Schulzrinne. An analysis of the Skype peer-to-peer Internet telephony protocol. Technical Report CUCS-039-04, Columbia University, 2004.
- [3] K. Borders and A. Prakash. Web tap: detecting covert web traffic. In *ACM Conference on Computer and Communications Security*, Oct. 2004.
- [4] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, IETF, Oct. 2004.
- [5] Dimes project. <http://www.netdimes.org/>, Aug. 2004.
- [6] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better NetFlow. In *Proceedings of ACM SIGCOMM 2004*, Portland, OR, Aug. 2004.
- [7] R. Huebsch, J. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *29th International Conference on Very Large Data Bases (VLDB '03)*, Sept. 2003.
- [8] G. Iannaccone, C. Diot, D. McAuley, A. Moore, I. Pratt, and L. Rizzo. The CoMo white paper. Technical Report IRC-TR-04-17, Intel Research, Sept. 2004.
- [9] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: Multilevel traffic classification in the dark. In *Proceedings of ACM SIGCOMM 2005*, Aug. 2005.
- [10] Microsoft. Event tracing. <http://msdn.microsoft.com/library/>, 2002. Platform SDK: Performance Monitoring, Event Tracing.
- [11] A. Mislove and P. Druschel. Providing administrative control and autonomy in structured peer-to-peer overlays. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, Feb. 2004.
- [12] A. Moore, J. Hall, E. Harris, C. Kreibech, and I. Pratt. Architecture of a network monitor. In *Proceedings of the Fourth Passive and Active Measurement Workshop (PAM 2003)*, Apr. 2003.
- [13] R. Mortier, R. Isaacs, and P. Barham. Anemone: using endsystems as a rich network management platform. Technical Report MSR-TR-2005-62, Microsoft Research, Cambridge, 7, JJ Thomson Ave, Cambridge, CB3 0FB. UK., May 2005.
- [14] Neti@home project. <http://www.neti.gatech.edu/>, Aug. 2004.
- [15] S. Saroiu, P. Gummadi, R. Dunn, S. Gribble, and H. Levy. An analysis of Internet content delivery systems. In *5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Dec. 2002.
- [16] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proceedings of ACM SIGCOMM 2004*, Sept. 2004.

Integrated Scientific Workflow Management for the Emulab Network Testbed

Eric Eide Leigh Stoller Tim Stack Juliana Freire Jay Lepreau

University of Utah, School of Computing

{eeide, stoller, stack, juliana, lepreau}@cs.utah.edu www.emulab.net

Abstract

The main forces that shaped current network testbeds were the needs for realism and scale. Now that several testbeds support large and complex experiments, management of experimentation processes and results has become more difficult and a barrier to high-quality systems research. The popularity of network testbeds means that new tools for managing experiment workflows, addressing the ready-made base of testbed users, can have important and significant impacts.

We are now evolving Emulab, our large and popular network testbed, to support experiments that are organized around scientific workflows. This paper summarizes the opportunities in this area, the new approaches we are taking, our implementation in progress, and the challenges in adapting scientific workflow concepts for testbed-based research. With our system, we expect to demonstrate that a network testbed with integrated scientific workflow management can be an important tool to aid research in networking and distributed systems.

1 Introduction

In the networking and distributed systems communities, research and development projects are increasingly dependent on evaluating real applications on *network testbeds*: sets of computers, devices, and other resources that either emulate real-world networks or are overlaid upon the real Internet. Data from testbed experiments are increasingly important as software systems become larger and more complex. Many testbeds are now available to researchers and educators worldwide, including PlanetLab, RON, the Open Network Laboratory, ORBIT, and Emulab [18].

Our research group has been developing and operating *Emulab*—a free-for-use, Web-accessible, time- and space-shared, reconfigurable network testbed—since April 2000. Emulab provides a common interface to a dozen types of networked devices, unifying them within a single experimental framework. These devices include

hundreds of (wired) PCs, a variety of wireless devices including PCs and motes, PlanetLab and RON machines scattered around the Internet, and emulated resources such as network conditioners and virtual machines. As of April 2006, our testbed has over 1,350 users from more than 200 institutions around the globe, and these users ran over 17,000 experiments in the last 12 months. Dozens of papers have been published based on experiments done with Emulab, and over 20 classes have used Emulab for coursework. In addition, Emulab's software is now operating more than a dozen other testbed sites around the world.

Emulab has continually grown to support larger and more varied experiments, and concurrently, users have applied Emulab to increasingly sophisticated studies. But how do experimenters manage their many experiments and their avalanche of data? Our answer is to evolve our system to support more structured experiments. The current interface to Emulab is based on a notion of isolated experiments: an experiment is essentially a description of a network topology that can be instantiated (emulated) on the testbed, along with optional event-driven activities. Many users have outgrown this model, however, and need better ways to organize, record, and analyze their work. Therefore, to support increasingly sophisticated and large scale research, we are evolving Emulab to support and manage scientific workflows *within* and *across* experiments.

A Different Domain, A Different Approach

We are inspired by the many scientific workflow management systems that have been developed for computational science in the Grid, including Kepler [10], Taverna [13], Triana [17] and others [19]. None of these apply directly to Emulab, however, because Emulab is about using unabstracted network resources. Moreover, Emulab's model of use is different, as described below.

It is fair to say that “general purpose” scientific workflow systems have been slow to catch on and are not yet in broad use [9, 14]. Anecdotal evidence indicates that resistance to their adoption stems from two main factors [9, 14]. First, a workflow system may be too con-

This material is based upon work supported by NSF under grants CNS-0524096, CNS-0335296, and CNS-0205702.

straining: the system does not match its users' needs in some manner, so users subvert the environment. Once workflow escapes the tool, a substantial part of the tool's benefits are lost. Second, the intended audience may lack "buy in." People are reluctant to change their work patterns to use a new tool. The impedance mismatch can be large or small, but even a small mismatch is significant. As traditionally approached and implemented, workflow management is about imposing discipline in ways that require active involvement by the user. Our experience with Emulab is that testbed users perform both structured *and* unstructured experiments, and that Emulab is most successful when it does *not* require special actions from its users.

Because we have tight control over both the testbed resources and the workflow user interface, we can do "better" than Grid workflow systems for our domain. To be successful, a workflow system must carefully fit into and support existing users' modes of use and the experiment life cycle, as the Taverna group describes [13]. We have experience with how people use Emulab and are crafting our workflow system to match. The following are some important ways in which our work is distinguished from other scientific workflow efforts, and we believe that these qualities will help us succeed.

- **Domain and Expertise.** We are tailoring our system to networking and distributed systems activities. That is precisely *our primary domain of expertise*, which we understand expertly. We have systems skills, helping us both to conceive and to execute techniques that most developers of workflow systems would not.

- **Experiment Model.** Emulab already has a pervasive *experiment* abstraction, which is a container of resources, control, and naming scope (Section 2). This abstraction eases both implementation and adoption: users are already familiar with it. Workflow can build on this abstraction and extend it in powerful but natural ways.

- **Implicit vs. Explicit Specification.** As an example of how our systems abilities benefit our workflow system, we can modify operating systems in the testbed and monitor the network to determine filesystem reads and writes. This helps us overcome the user-adoption barrier in which the experimenter must exercise discipline by explicitly specifying all input and output parameters. We can instead *infer* that opened files are parameters, and simply record all file state, or provide users with straightforward options to deal with those files (Section 3).

- **History-Based Views.** A key part of our philosophy is to "remember everything." We can leverage the trend in cheap disk space and copy-on-write filesystem techniques to (in principle) record everything that occurs within an experiment. One possible implementation technique is to use a filesystem that supports snapshots, such as ZFS [16], or new research storage systems that

support branching [1, 15], to provide immutable storage efficiently. We provide a simple GUI for users to peruse and branch the "tree" of experiment history (Section 4).

- **Incremental Adoption.** Researchers and students already make heavy use of the Emulab testbed. We build on that to transparently add features under the covers, lure users to exploit them, and get explicit and implicit user feedback. Emulab's interface is Web-based, so like all ASPs we can track experimenters' use of features and their paths through our interface.

- **Pragmatic Approach.** Finally, we have adopted a thoroughly practical attitude toward the incorporation of scientific workflow into the user experience of our testbed. As we incorporate features to support workflows within and across testbed experiments, we strive to maintain high degrees of transparency, flexibility, and extensibility. Our goal is for users to be able to choose the degree to which they want to interact with workflow features, and change their minds as their activities require.

Our "bottom-up" strategy for integrating scientific workflow into Emulab is designed so that our system will be practical for the users of our testbed. Beyond implementing a useful system, we are devising new techniques to enable the "bottom-up" design and adoption of workflows for our domain. We are hopeful that the techniques we develop and the experience we gain will be applicable beyond the systems and network areas.

Contributions

Our work addresses a convergence of demand and opportunity, and Emulab is a uniquely powerful environment in which a workflow system can succeed. The testbed provides leverage such as the ability to repeat distributed system experiments "exactly" and collect data automatically. We are using *implicit* and *automatic* techniques for workflow definition and execution where possible, and monitoring for *completeness*. We have much history on Emulab's use and are guided by actual user experience (including our own) which shows the need for *flexible* user interactions and patterns for workflow *history* and *groups*. We are deploying features in an *evolutionary* way, thus helping users and enabling feedback. Maintaining a *low barrier to entry* is a major goal of our work. The rest of this paper describes the opportunities and challenges ahead, and summarizes the status of our workflow-integrated Emulab in progress.

2 Testbed Experiments

In the current Emulab, the main conceptual entity is an *experiment*. Primarily, an experiment describes a network: a set of computers and other devices, the network links between them, and the configuration of those

devices and links. An experiment defines such things as the hardware type of each node in the network (e.g., a 3 GHz PC or a mobile robot), the operating systems and software packages to be loaded on those nodes, the characteristics of each network link (e.g., speed, latency, and loss), and the configuration of other entities such as network traffic generators. A user describes these items in an extension of the *ns* language [8], or alternatively, through a GUI in Emulab's Web interface. Either way, the user submits the description to Emulab and gives it a name. Emulab parses the description into an internal form and stores the data in its database.

At this point, the experiment can be “swapped in.” When a user directs Emulab to swap in an experiment, Emulab maps the logical topology of the experiment onto actual testbed hardware, loads the requested operating systems and software onto the allocated devices, creates the network links (using VLANs), and so on.

When swap-in is complete, the user can login to the allocated machines and do his or her work. In contrast to swap-in, which is handled by Emulab, the organization and execution of the actual experiment steps are mostly left up to the user. Many users perform their experiments interactively, via remote shell access to the allocated computers. More sophisticated users, however, often automate some or all of the steps. Emulab provides some tools for this—e.g., an event service—but users must provide the higher-level frameworks and orchestration that make use of Emulab's built-in tools.

A user's experiment may last from a few minutes to many weeks, allowing the user time to run multiple tests, change software and parameters, or do long-term data gathering. When the user is done, he or she tells Emulab to “swap out” the experiment, releasing the experiment's resources. The experiment itself remains in Emulab's database so it can be swapped in again later.

3 Workflow Within Experiments

It has become clear that Emulab's experiment notion must change to meet the needs of increasingly complex experimental activities. Three critical requirements are *encapsulation*, *orchestration*, and *data management*, and satisfying these needs is the job of a scientific workflow system. Consequently, we are now evolving Emulab's interfaces and internals to support experiments built around workflows. Organizing Emulab experiments around existing scientific workflow metaphors and mechanisms is not straightforward. Here, we outline some of the challenges and the ways we are extending existing workflow notions to meet our users' needs.

Encapsulation. An essential part of a workflow system is keeping track of artifacts, i.e., experiment inputs and outputs. Unfortunately, Emulab's current model of

an experiment does not encapsulate all the artifacts that are part of the experimental process. Some constituents, such as the software packages installed at swap-in, are referenced by name only. The experiment definition does not contain the software, but instead contains only a file name. Other objects including command scripts, program input files, and especially program output files, may not be referenced at all in an experiment description. Users simply know where these objects are located in Emulab's file system.

A critical first step toward a workflow-enabled testbed, therefore, is to evolve Emulab's representation of experiments to encapsulate all the elements—insofar as possible—that make up an experiment. This evolution has many technical aspects, and Emulab already provides a solid base for addressing them. For instance, Emulab today encapsulates disk images [7], which contain operating systems and other software. These are currently referenced from experiments by name, but it will be “straightforward” to keep disk images within an expanded type of experiment.

The more challenging aspects of encapsulation concern the user interface, and the effort required for a user to create an encapsulated experiment. It is important for the set of experiment constituents to be *complete*, so an experiment can be repeated and modified; it must also be *precise* (without extraneous detail), so the experiment can be stored, analyzed by the testbed, and understood by people. Because even a small experiment may read and/or write hundreds of files, it would be unworkable for Emulab to require users to identify all the parts of an experiment by hand.

Therefore, we are using *automatic approaches* to determine the “extent” of experiments. To capture file data, we have modified Emulab to snapshot all the files referenced by an experiment, every time the experiment is swapped in or out. In general, not all of these files will be named in an experiment's definition. To find files that are not named there (e.g., output files), we have implemented dynamic monitoring of filesystem accesses. Snapshots are persistent and thus provide a sound basis for encapsulating experiments and tracking revisions over time. We may supplement or replace our custom tracking solutions with a storage system that is “provenance-aware” [11]. Such a system tracks the lineage of files and supports queries over files' histories. Our workflow system may be able to use such a system to dynamically discover or verify the commands that were used to create files within an experiment.

Dynamic techniques may not be precise, however, in the sense that they may capture irrelevant detail. As our work evolves, we expect to combine automatic and manual techniques for defining experiments precisely: automatic for producing candidates and checking encapsu-

lation, and manual for precision, classification, and annotation. For instance, at the end of an experiment, the Emulab GUI can present a list of files that were accessed but that were not previously defined to be “part of” or “not part of” the experiment. The user may choose to classify the files, and thereby improve the description of the experiment. If the user chooses not to classify the files, however, default behaviors would apply. The PASS described previously will provide the necessary metadata to make these defaults intelligent.

Orchestration. Emulab is a platform for directed experimentation, but just as important, it is a platform for open exploration. Emulab configures computers and networks very quickly (within minutes) and provides users with interactive access and total control over their machines. For these reasons, Emulab is often used for ad hoc activities including software development and debugging, platform testing (e.g., with many different operating systems), and live demonstrations.

Therefore, a major focus of our workflow integration is to accommodate the wide variety of ways in which Emulab is used: for directed and exploratory work, with interactive and scripted commands, and with workflows that blend and combine these modes. The practical issue is for the workflow facilities not to get in the way when they are not wanted. The research issue is how to meet that goal while also supporting the evolution of unplanned and/or interactive experiments into repeatable and/or scripted workflows. This capability will necessarily build on Emulab’s ability to monitor essentially all activity within the testbed. Using an automated workflow in an interactive way is also essential: a user may want to execute the workflow up to a point, and then “take the wheel” to explore a new idea (Section 4).

Currently, due to the effort required, relatively few Emulab users take the time to carefully script and package their experiments. By evolving the current experimentation model, we expect to lower the barrier to orchestrating Emulab experiments, expand the set of activities that can be coordinated, and provide a framework in which users can create new types of workflow steps. By integrating a workflow system and libraries of workflow elements, we will make it easier to create experiments from reusable “parts.” For example, we have prototyped a point-and-click interface that generates graphs from data stored in a broad range of formats. We will pursue similar ideas, based on programming-by-example (i.e., recording the user’s actions), to make it easier for users to interactively develop and record their workflows.

Data Management. Pre-packaged workflow elements help us present other new capabilities to users as well: e.g., automated collection of experiment data (via probes), data analysis steps, and visualization steps. For these tasks, we and our collaborators are working

to connect a *datapository* [3]—a network measurement data storage and analysis facility—to Emulab. Experiment measurements will be saved in the datapository’s database. By structuring user interactions as workflows, we will be able to describe data analysis and visualization steps that can occur after the primary resources for an experiment run have been released, i.e., after “swap out.” An important issue will be to expand Emulab’s resource scheduling capabilities: e.g., to handle demands that change over the course of a workflow and to handle new types of resources such as CPU and I/O within the datapository.

4 Workflow Across Experiments

The thorough study of a system typically requires many experiments. For example, an Emulab user may want to study a distributed system across a variety of network sizes and topologies. Furthermore, as results are obtained from experiment runs, the user may change his or her plan for future experiments. Users routinely modify experiment definitions to explore promising new avenues as they are discovered—and then want to return to the original course, or explore yet another new direction. Thus, testbed users need to navigate easily through “experiment space” on both planned and unplanned courses. As opposed to managing workflow *within* a single experiment, this type of navigation corresponds to workflow *across* experiments. Workflow across experiments deals with flexible grouping, managing changes to experiment definitions, and navigating through artifacts across time.

Definition and Execution. An essential first step is separating the main temporal aspects of experiments: i.e., definition and execution. These aspects are currently intertwined: for example, an experiment is either swapped in or swapped out, and a user cannot run two copies of a single experiment at the same time. These are properties of the design described in Section 2, which many Emulab users have outgrown.

As we evolve experiments to be more encapsulating, we are adding the ability to track and distinguish results that are produced by each run of an experiment. This is similar to the separation of workflow definition and execution found in the VisTrails system [4]. Even if an experiment description does not change, its output may—for example, due to the impacts of real-world effects such as the state of the actual Internet. The many runs of a single experiment form a group, and Emulab will provide users with straightforward access to the members of such groups. For example, users will be able to see if and how the results of an experiment change over time.

Grouping. Beyond grouping the executions of a single, unchanging experiment, our experience is that users need much more sophisticated grouping facilities. Most

importantly, users need flexible ways to manage groups of experiment *definitions* in addition to groups of *executions*. A user who wants to run a collection of related experiments today must do so in an ad hoc manner: by creating many similar but “unrelated” experiments, by modifying one experiment many times, or by combining all the tests into a single Emulab experiment. All of these approaches, however, disguise the essence of the collection from Emulab. Support for workflows over groups of experiments, therefore, will require Emulab to have a built-in notion of experiment groups. We are currently prototyping support for experiment groups in Emulab, which involves two main challenge areas.

First, *parameterized experiment definitions* capture a common kind of grouping but present new user interface and scheduling issues. A parameterized definition is like a subroutine with formal parameters: these are bound to values when the experiment is executed. A parameterized definition is therefore a *template* of experiments to be run. We have prototyped a user interface for this in Emulab, in which a user specifies parameter values in a Web form. This will be enhanced so users can save and reuse parameter sets. It will also be extended so users can describe *movement* through the parameter space of an experiment. By creating workflows that move through parameter space, Emulab will provide a scalable way to execute large numbers of trials. This presents a scheduling issue: the ability to explore parameter space leads to new concerns such as the desire to get “interesting” results quickly, where “interesting” is determined by system- or user-defined metrics. Our work in this area will build on adaptation features that are already in Emulab [6]. Smart scheduling highlights the need for workflow to be *integrated* with Emulab, not merely built on top.

Second, although parameterized definitions produce groups of related experiment runs and results, it is essential for “group” to be a more flexible concept, allowing users to define and navigate through *arbitrary* sets of experiments and results. Users will want to group experiment results in many ways, none of them necessarily being inherent or dominant. One experiment may belong to many groups at the same time: for instance, a test of a Web server may belong to a series in which the client load varies (requests/sec) and to one in which the server configuration varies (e.g., number of threads, or caching strategy). Users will want to define groups by extension (e.g., the records of particular experiment executions), intension (e.g., rules that select records of interest), and combinations of these. New groupings will be created after experiments are run, and parameters may be added or removed over time. All of these things mean that we must develop a highly flexible means for cataloging and tracking experiment records.

History. User-defined experiment groups support

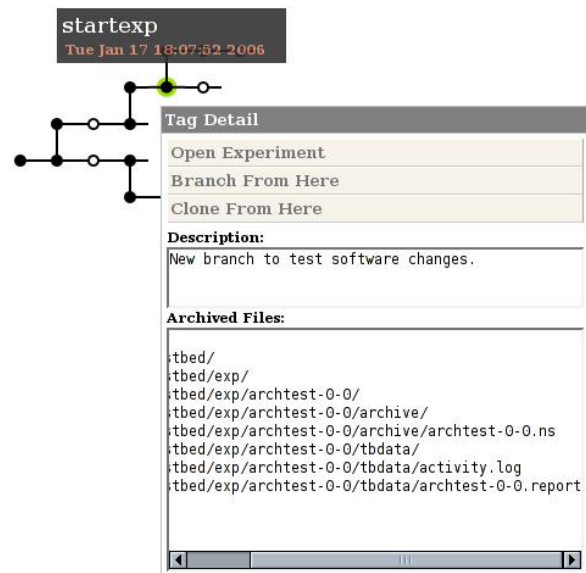


Figure 1: Screen shot of our prototype Web-based browser and editor of the experiment tree archive.

“spatial” navigation across related experiments. Just as important, however, is temporal navigation over experiments, singly or in groups. As previously described, users routinely modify their experiments over time to explore both planned and unplanned directions of research. We have found that it is important for users to be free to make such changes without fear of losing previous work; to name and annotate experiment versions; and to “back up” to previous versions of an experiment, and from there “fork” to explore new avenues.

The revisions of an experiment form a tree, which represents the provenance of an experiment workflow. We have implemented a prototype of experiment trees in Emulab, based on the Subversion revision control system and an AJAX-based GUI, integrated into Emulab’s Web interface (Figure 1). Previous work by Bavoli et al. [4] and Callahan et al. [5] provides additional detail about managing workflow histories in general.

5 Related Work

There is growing interest in applying scientific workflow systems to testbed-based research. For example, Plush [2] is a workflow-centered tool for experiments within PlanetLab. Plush focuses on tool integration and experiment automation; in contrast, our workflow enhancements to Emulab focus on strong encapsulation (including repeatability) and enhanced exploration (including flexible use, groups, and history management). Both Plush and our work aim to support extensions via third-party and user-provided workflow steps.

LabVIEW [12] is a popular commercial product for scientific experiment management and control. It is similar to our Emulab-based workflow system in that both manage workflow processes and results from “instruments.” However, the domains of the two systems differ in two ways. First, the instruments in Emulab consume and produce many complex sources of data: e.g., configurations of hardware and software, input and output files and databases, software under test (sources and binaries), and data from previous runs and variations of an experiment. Emulab and its users must deal with a wide variety of highly structured data, not just series of sensor readings. Second, Emulab is the laboratory, not just the monitoring and control portion of it. Our integrated workflow system has near-total control over the relevant components of the environment in which Emulab experiments occur, so it can perform tasks such as setting up hardware or virtual machines, and running experiments automatically, with probes in place. Furthermore, to a significant extent, our system can archive the actual “devices” under test, not just the recorded outputs of those devices. The ability to archive and re-execute software—including entire disk images—means that we can provide much more automation in our domain than what LabVIEW can generally provide in its domain.

6 Conclusion

As grids changed the playing field for computational science, testbeds like PlanetLab and Emulab have changed the field for networked and distributed systems. The scale, complexity, and popularity of network testbeds have reached the point where scientific workflow systems are often needed to manage testbed-based research. To meet this need, we are integrating novel workflow support in Emulab, both within and across experiments. Moreover, testbeds like Emulab are an opportunity for advancing scientific workflow systems themselves. By building on and retaining Emulab’s strengths, including “total” experiment monitoring and interactivity, we are expanding the domain of scientific workflow, developing new workflow and experiment management techniques, and, we predict, achieving new levels of acceptance and adoption for scientific workflow systems in general.

Acknowledgments

We thank Mike Hibler, Russ Fish, and the anonymous reviewers for their valuable comments on this paper.

References

[1] M. K. Aguilera, S. Spence, and A. Veitch. Olive: Distributed point-in-time branching storage for real systems. In *Proc. Third NSDI*, May 2006.

[2] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. PlanetLab application management using Plush. *ACM Operating Systems Review*, 40(1):33–40, Jan. 2006.

[3] D. G. Andersen and N. Feamster. Challenges and opportunities in Internet data mining. Technical Report CMU-PDL-06-102, Carnegie Mellon University Parallel Data Laboratory, Jan. 2006. www.datapositionary.net.

[4] L. Bavoli, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: Enabling interactive multiple-view visualizations. In *Proc. IEEE Visualization 2005*, pages 135–142, Oct. 2005.

[5] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Managing the evolution of dataflows with VisTrails. *IEEE Workshop on Workflow and Data Flow for Scientific Applications (SciFlow 2006)*, Apr. 2006. Extended abstract. <http://www.cs.utah.edu/~juliana/pub/sciflow2006.pdf>.

[6] M. Hibler et al. Feedback-directed virtualization techniques for scalable network experimentation. Flux Technical Note FTN-2004-02, University of Utah, May 2004. <http://www.cs.utah.edu/flux/papers/virt-ftn2004-02.pdf>.

[7] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, scalable disk imaging with Frisbee. In *Proc. 2003 USENIX Annual Tech. Conf.*, pages 283–296, June 2003.

[8] ISI, University of Southern California. The network simulator - ns-2. <http://www.isi.edu/nsnam/ns/>.

[9] G. Lindstrom. Personal communication, 2005–2006.

[10] B. Ludäscher et al. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, Dec. 2005.

[11] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. 2006 USENIX Annual Tech. Conf.*, June 2006.

[12] National Instruments. LabVIEW home page. <http://www.ni.com/labview/>.

[13] T. Oinn et al. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, Dec. 2005.

[14] S. Parker. Personal communication, Jan. 2006.

[15] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proc. Third NSDI*, May 2006.

[16] Sun Microsystems, Inc. ZFS home page. <http://www.opensolaris.org/os/community/zfs/>.

[17] I. Taylor, I. Wang, M. Shields, and S. Majithia. Distributed computing with Triana on the Grid. *Concurrency and Computation: Practice and Experience*, 17(9):1197–1214, Aug. 2005.

[18] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th Symp. on Operating Systems Design and Impl.*, pages 255–270, Dec. 2002.

[19] J. Yu and R. Buyya. A taxonomy of workflow management systems for Grid computing. Technical Report GRIDS-TR-2005-1, Grid Computing and Distributed Systems Laboratory, Univ. of Melbourne, Mar. 2005.

How DNS Misnaming Distorts Internet Topology Mapping

Ming Zhang
Microsoft Research

Yaoping Ruan
IBM Research

Vivek Pai, Jennifer Rexford
Princeton University

Abstract

Network researchers commonly use reverse DNS lookups of router names to provide geographic or topological information that would otherwise be difficult to obtain. By systematically examining a large ISP, we find that some of these names are incorrect. We develop techniques to automatically identify these misnamings, and determine the actual locations, which we validate against the configuration of the ISP's routers. While the actual number of misnamings is small, these errors induce a large number of false links in the inferred connectivity graph. We also measure the effects on path inflation, and find that the misnamings make path inflation and routing problems appear much worse than they actually are.

1 Introduction

Network researchers commonly use reverse DNS lookups to infer router locations when extracting network topology and routing behavior, since large ISPs often embed topological or geographic information in the router's DNS name. Using these names, outsiders can infer information that would otherwise require explicit cooperation from the ISP. For example, decoding all of the router names seen during a traceroute can show what cities a network path visits. Previous research has used the information inferred from this approach to map ISP topology [12] and estimate network distance [4, 11, 13].

While this technique has been shown to be useful, errors can occur for a variety of reasons, affecting the conclusions drawn from this data. Router interfaces are often given DNS names manually by network operators, for troubleshooting convenience rather than as a primary addressing mechanism. As routers and line cards are moved, reconfigured, or cycled out of service for repairs or upgrades, and as IP addresses are reassigned across the ISP's network, the DNS information may not be updated, and inferences drawn from it become inaccurate. These naming errors may persist for long periods if they have no effect on normal network operations—the network operators may never need to perform troubleshooting on the incorrectly-named interfaces. However, external researchers attempting to analyze the ISP's network may be affected by these misnamed interfaces.

Without correcting for these DNS misnamings, researchers may get misleading or even conflicting re-

sults when applying inference techniques based on DNS names. We are unaware of any examination of the errors in this approach and their implications. In this work, we present the first systematic study on DNS misnamings, with validated results. Our contributions are as follow:

- We propose ways to detect misnamings, based on observing “abnormal” paths via traceroute. For example, we find stable paths that appear to visit the same point-of-presence (POP) multiple times.
- We develop heuristics for identifying and fixing misnamed addresses by correlating traceroutes from multiple vantage points. We analyze a large ISP and validate against the ISP's router configuration data.
- We examine the topological impact of DNS misnamings. Although DNS misnamings only occur in a small portion (0.5%) of IP addresses, their topological impact is disproportionately larger—we find that 20 out of 182 (11%) edges in a Rocketfuel-like network topology [12] are actually false edges.
- We find that DNS misnaming has an even greater impact on path inflation. Correcting the misnamed addresses reduces the number of unusually long paths by more than 50%.

In the rest of this paper, we describe the system we developed to map the ISP, how we find and resolve the naming problems, and how we determine the impact of these problems on the topology and routing measurements. We have performed these measurements on one large ISP, and have verified with them that the misnamings exist and that our solutions are correct.

2 Inferring POP-Level ISP Topologies

To understand how DNS misnaming affects researchers, we discuss how modern ISP networks are constructed, and what complicates the process of inferring their topology. At a high level, an ISP's network is a set of cities that have Points-of-Presence (POPs), and the links that connect these POPs. The POPs contains the routers that connect the ISP's links, and may also provide easy access to links of other peer ISPs and customers. These routers have multiple interfaces with separate IP address, and may also have DNS names configured for reverse lookups. A POP may also have multiple interconnected routers, rather than a single, larger router.

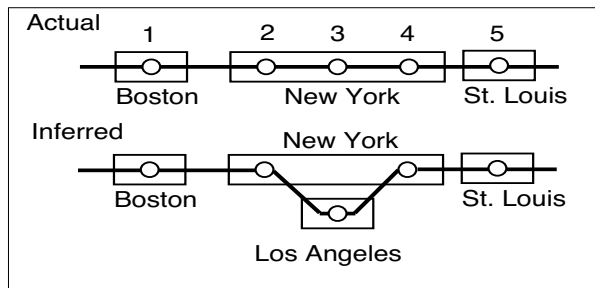


Figure 1: Mismatching causing a POP loop and extra edges. Circles are routers, and rectangles are POPs.

Tools like `traceroute` only report the IP addresses of the interfaces on the forwarding path, but not the POPs traversed. To derive POP-level topology, the interface IP addresses must be mapped to their corresponding POPs. While the network operators have this information readily available to them, external researchers do not, and must use some other means to infer it.

The commonly-used mapping method is to perform reverse DNS lookups on the returned IP addresses, and then extract the city name or POP code that many large ISPs embed in the DNS name. For example, 12.122.12.109 reverse-resolves to *thr2-p012601.phlpa.ip.att.net*, indicating it is an AT&T router in Philadelphia (phlpa), and 144.232.7.42 reverse-resolves to *sl-bb22-nyc-6-0.sprintlink.net*, indicating it is a Sprint router in New York City (nyc). By mapping from IP addresses to POPs, researchers can then extract other information, such as what cities are visited along a path and how many routers are traversed in each POP.

DNS misnaming can cause severe errors in inferred topologies, as shown in Figures 1 and 2. In Figure 1, the actual path has one router in Boston and three routers in a POP in New York. The inferred topology has a POP loop because the DNS name of *IP3* is misconfigured with a name that suggests the interface is located in Los Angeles. Figure 2 shows a simple topology consisting of many routers in a large POP in San Francisco, with connections to Seattle and Salt Lake City. Reverse DNS lookup of *IP3* suggests the router is within Seattle while it is actually in San Francisco. DNS misnaming causes four major effects on topology inference:

- **Path inflation:** In Figure 1, the misnamed router induces the POP-level “loop,” making the path appear needlessly inflated, since the Los Angeles round-trip is unnecessary. The effect on inferred path inflation can be severe, particularly for short paths.
- **False edges:** If the NYC POP in Figure 1 does not have any real links to LA, the misnamed router suggests that these POPs are directly connected, adding a false edge to the inferred topology.

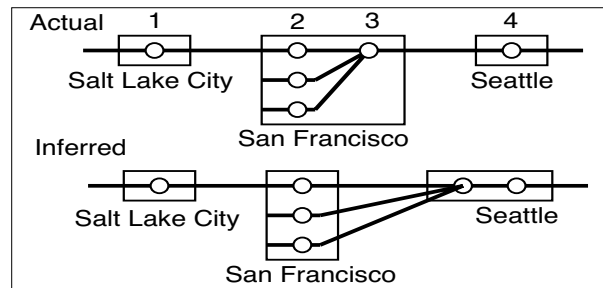


Figure 2: Mismatching can shift routers across POPs, yielding multiple edges between a pair of POPs

- **Extra inter-POP links:** In Figure 2, both ends of the SF–Seattle link are labeled as being in Seattle, causing the dense intra-POP links in SF to appear as multiple links to Seattle. Though technically possible, such redundant links are unlikely, since a smaller number of higher-capacity links would require less hardware and less expense.
- **Missing edges:** If router 3 in Figure 2 were misnamed as another city, such as Los Angeles, then the traceroute path would not contain a direct SF–Seattle connection, causing the inferred topology to miss a real link between the two POPs.

3 Data Collection

To map the ISP topology, we perform distributed traceroutes that traverse many paths of the network under study. The reason for distributed traceroutes is not only to improve coverage of the ISP’s links, but also to view mislabeled IP addresses from multiple vantage points.

3.1 Traceroute Measurements

We perform traceroutes from 132 nodes on Planet-Lab [7], across sites in the US, Canada, South America, Europe, Middle East, and Asia. From each node, we perform traceroutes to all 265,448 prefixes in the BGP tables of RouteViews [5], RIPE-NCC [8], and Route-Server [9]. Some of these prefixes are either partially or completely superseded by more specific subnets. To discard these prefixes, we use the algorithm from Mao *et al.* [3] to extract 259,343 routable address blocks. We randomly pick one destination IP address in each block to traceroute and we remove unstable paths caused by routing changes. We modify traceroute to probe only a single destination port to reduce the chance of being accused of port scanning. We also use a blacklist to avoid known prefixes that easily trigger alarms. Data collection spanned 20 hours on March 30, 2005.

To study the misnaming of a specific ISP, we first pick the traceroutes that traverse the target ISP. We use the

BGP tables to map IP addresses to their autonomous systems (ASes). The mapping is constructed by inspecting the last AS, termed the *origin AS*, in the AS path for each prefix [1]. Some IP addresses may map to multiple origin ASes (MOAS) [15], in which case we consider it part of the target ISP if one of the origin ASes is that ISP. With the IP-to-AS mapping, we can then identify all the traceroutes that intersect with the target ISP.

3.2 IP-to-POP Mapping

To obtain POP-level information, we perform the reverse DNS lookups of the IP addresses encountered by traceroute, and then use the parsing rules of the *undns* tool [10] to extract POP-level information. Version 0.1.27 of *undns* has parsing rules for 247 ASes. For our target AS, we added four new city names for POP names that were not present in *undns*.

With the POP-level information of an IP, we use the longitude and latitude of the city as an estimate of the geographic location of that POP. We acquire the geographic location through Yahoo maps, by requesting a map of the city/state pair; the latitude and longitude of the city are embedded in the HTTP response. This enables us to calculate the geographic distance between two POPs. We will discuss this in more detail in Section 5.3, where we quantify the impact of misnaming on path inflation.

4 Identifying and Correcting Misnamings

In this section, we present our algorithms for identifying misnamed router interfaces and associating them with the correct POPs. We propose two heuristics for detecting and correcting misnamed interfaces.

4.1 POP-Level Loop

Normally, a path inside an ISP should not contain a POP-level loop, because ASes typically employ intradomain routing protocols that compute shortest paths using link weights. Inter-POP link weights are usually much larger than those of intra-POP links, to reduce propagation delay and avoid overloading expensive long-haul links. Therefore, for stable paths, the traffic that passes through a POP should not return to the same POP again.

To determine which IP address in a POP-level loop has been mislabeled, we leverage our distributed traceroutes. Misnamed IPs are likely to appear repeatedly in the abnormal paths when we combine the traceroutes from multiple locations. Assuming we have a collection of stable paths with POP-level loops, a simple strategy is to count how many times each IP appears and pick the ones that appear most frequently. However, this strategy may not work well, because it treats all the IPs equally. For example, a correctly-named IP address may appear frequently, simply because it is close to a misnamed IP.

To handle this problem, we assume that most DNS entries are correct and misnamings are infrequent, which we see is true for the ISP we study in Section 5. Therefore, we could resolve all the POP-level loops by fixing only a small number of misnamed IP addresses. We devise a greedy algorithm to solve this problem.

For each abnormal path with a POP-level loop, we have several possible candidates for misnamings. For each interface in the path, we check if we can resolve the loop by mapping this address to a different POP. If so, we consider this IP possibly misnamed. For example, in the inferred path in Figure 1, the second and the fourth IPs are candidates, since we can break the loop by mapping either of them to the Los Angeles POP. The third IP is also a candidate, because we can resolve the loop by mapping it to New York. In this way, we can obtain a set of candidate misnamings for each abnormal path. To select the most likely candidate, we consider all abnormal paths together. Our goal is to identify the minimum set of IPs that needs to be relabeled to resolve all the loops.

The pseudocode of our greedy algorithm for identifying misnamed IPs is shown below. We first compute the candidate set for each abnormal path. Then we greedily pick a candidate IP address that helps to resolve loops for many paths, while at the same time it seldom appears in a path where renaming does not resolve its loop. Finally, we remove the paths whose loops can be resolved by the selected IP and output this IP. This process continues until there are no abnormal paths.

```

For each abnormal path
  Compute the candidate set of misnamed IPs;
While the set of abnormal paths is not empty
  Compute the union of all candidate sets;
  For each candidate IP in the union set
    Count the # of paths where it is in
      their candidate set, CountCandidate;
    Count the # of paths where it appears
      but not in their candidate set,
      CountNotCandidate;
  Pick CandidateIP with the max value of
    CountCandidate - CountNotCandidate;
  Remove all the abnormal paths whose loop
    can be resolved by fixing CandidateIP;
  Output CandidateIP;

```

After identifying the misnamed IP addresses, the next question we want to answer is: can we find the correct POPs of those misnamed IPs by only examining the traceroute data? If so, we can then resolve the misnamings without the ISP's internal data, and supplement the existing topology mapping systems with this DNS name auto-correcting mechanism to achieve higher accuracy.

As we just described, we test if we can resolve a loop by mapping an IP to a different POP. We often have multiple choices—for example in Figure 1, we can map IP_4 to Los Angeles, St. Louis, or any other POP that does not appear in the path to resolve the loop. However, IP_4

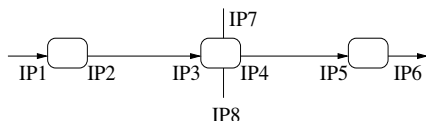


Figure 3: Misnaming leads to router-level discrepancy.

is more likely to be in Los Angeles or St. Louis than in some other random POP because it is connected to both POPs. Therefore, we assign a misnamed IP to a POP by voting based on its neighbors [2]. If the majority of them map to the same POP, we consider it the correct POP for that IP. We assume that routers have more intra-POP links than inter-POP links. Given that inter-POP links span much longer distances and are more expensive, we believe this assumption is true for most major ISPs.

4.2 Router-Level Discrepancy

Traceroute usually reports the IP address of the incoming interface of each router on the forwarding path. For example in Figure 3, the traceroute only reports IP_1 , IP_3 , and IP_5 along the path. Sometimes, we can infer the IP of the outgoing interfaces from that of the incoming interfaces. We take advantage of the fact that the inter-POP links of many major ISPs are high-speed point-to-point links (e.g., Packet-Over-SONET links). This means the IP addresses at the opposite ends of a link are in the same /30 subnet. Among the four IPs in a /30 subnet, the two ending with 01 and 10 are used as router interface addresses while the two ending with 00 and 11 are used as network and broadcast addresses respectively. So, if we know that both IP_3 and IP_5 (144.232.9.149) are backbone routers, we can infer that IP_4 is 144.232.9.150 and obtain its DNS name. Since IP_3 and IP_4 are on the same router, their names should map to the same POP; if not, we call this a *router-level discrepancy*.

We collect all such abnormal IP pairs and assign each IP address to a router. For example, in Figure 3, suppose there are three such pairs, (IP_3, IP_4) , (IP_3, IP_7) , and (IP_3, IP_8) . We will assign IP_3 , IP_4 , IP_7 , and IP_8 to the same router. Then for each router, we decide its correct POP by voting. If the majority of its interfaces map to the same POP, we consider it the correct POP of that router, and the IP that maps to a different POP a misnamed IP. For example, suppose IP_4 , IP_7 , and IP_8 map to Chicago while IP_3 maps to Detroit, we infer that IP_3 is misnamed and it should map to Chicago.

This heuristic may not work if a router is moved to another POP with none of the DNS names of its interfaces being updated. In practice we have never seen such a case. However, even if this case does occur, it will be most likely to be detected by POP-level loops since there will be many misnamed interfaces. We can resolve it by voting based on its neighbors as described in Section 4.1.

IP	Wrong POP	Correct POP	Method
1	WA	CA	Loop
2	MA	CO	Loop
3	FL	CO	Loop
4	CA	CO	Loop
5	VA	DC	01/10
6	VA	DC	01/10
7	City A, CA	City B, CA	Missed
8	City A, CA	City B, CA	Missed
9	City C, PA	City D, PA	Missed

Table 1: Summary of all misnamed IPs. Loop: POP-Level Loop, 01/10: Router-Level Discrepancy

We could have also used the IP alias check [12] to detect misnamed IPs, but we may not know the right IP pairs to compare in advance. For example, in Figure 3, IP_7 , IP_8 , and IP_4 may not appear in the traceroute measurements without using the 01/10 rule. Even if they do appear, we may not know to check IP aliases between IP_3 and $IP_7/IP_4/IP_8$ because their DNS names look unrelated. The 01/10 rule helps us to quickly identify a small number of abnormal IP pairs and focus on them.

5 Case Study on a Large ISP

In this section, we validate our algorithms for identifying and fixing misnamed IPs against the router configuration data for a large ISP. We then study the impact of misnamed interfaces on the inferred topology and path inflation. Although ISPs' naming conventions may be different, the techniques we describe are applicable to other ISPs as well. We plan to study other ISPs in the future.

5.1 Validation With Configuration Data

The ISP under study (kept anonymous by agreement) has hundreds of routers and dozens of POPs at different cities around the United States. We first select the traceroutes that traverse the ISP. As described in Section 3.1, we traced to 265,448 prefixes from 132 nodes on PlanetLab. After applying the IP-to-POP mapping, we discovered 113 POPs, which cover most of the ISP's POPs.

Among the traceroutes that traverse the ISP, we find 1,957 paths with non-transient POP-level loops. Using the algorithm described in Section 4.1, we are able to identify four misnamed IPs, which are listed as IP_1 , IP_2 , IP_3 , and IP_4 in Table 1. By comparing with the router configuration data, we confirm that these four IP addresses are indeed misnamed. In addition, the voting algorithm in Section 4.1 is able to map those misnamed interfaces to their correct POPs.

Since the ISP is a large backbone provider, most internal links are point-to-point links. We use the router-level discrepancy heuristic described in Section 4.2 to look for

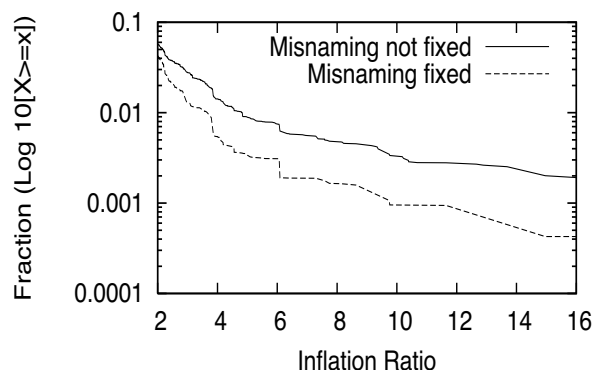


Figure 4: CCDF of path inflation ratio before and after fixing misnamings.

misnamed interfaces in all the non-transient traceroute results. This heuristic allows us to identify two more misnamed interfaces— IP_5 and IP_6 in Table 1. We again confirm that these interfaces are misnamed and that our voting algorithm maps them to the correct POPs.

Finally, we check the completeness of our algorithms. Although we are able to identify six misnamed IPs, we fail to detect three misnamings, which are IP_7 , IP_8 , and IP_9 in Table 1. A closer look at the traceroute data reveals that each of the three IPs has only one neighboring POP and is misnamed to its neighboring POP. IP_9 actually resides in City D , which is a nearby suburb of the larger City C in its name; similarly, IP_7 and IP_8 are located in a small City B near a large POP in City A in California. There is no way that we can identify these misnamed interfaces based on traceroute measurements. Arguably, this type of misnaming has very limited impact on topology mapping and path inflation, since these are small POPs with a degree of 1 and are misnamed as a big POP that is very nearby.

5.2 Impact on Topology Mapping

As discussed earlier in Section 2, misnamed interfaces may lead to false edges in topology mapping. Using the mapping techniques in [12], we find that the six misnamed interfaces (IP_1 to IP_6 in Table 1) lead to *twenty* false edges which do not exist in the real topology. This corresponds to 11% of the total number of inferred edges. We can see that although misnamed IPs are rare, they have a significant influence on topology inference.

Past work relies on the speed-of-light rule to identify false edges [12]. To determine whether a link is false, we first infer the geographic location of the two endpoints of the link from their DNS names. Based on this information, we calculate the shortest time it takes for light to traverse the distance between the two endpoints. Then we estimate the one-way latency of the link using the actual RTT measurements in traceroute. If the latency

estimated from the traceroute is smaller, we know the inferred location of at least one of the endpoints is wrong and the link is false. However, the speed-of-light rule has some limitations. First, it can only identify false edges whose actual distance is shorter than the distance inferred from DNS names. Second, the one-way link latency estimated from traceroute measurements may be inaccurate because of Internet routing asymmetry, queueing delay, or delay in router response. Third, it can only detect misnamed IPs but cannot assign the IPs to the correct POPs. In our dataset, the speed-of-light rule only identifies 1 misnamed IP. In comparison, we discover and fix 6 out of the 9 misnamed IP addresses.

5.3 Impact on Path Inflation Studies

Misnamed IPs may inflate the linearized geographic distance of a path, as we explained briefly in Section 2. We now study to what extent misnamings may affect path inflation. As in [13], we compute the *inflation ratio* of a path as the ratio of the linearized distance of a path to the geographic distance between the source and the destination. This ratio reflects how much a path is inflated because of network topology constraints [11].

We calculate the inflation ratio for every possible IP-level path inside the ISP. Figure 4 compares the complementary cumulative distribution function (CCDF) of the inflation ratios, before and after correcting the misnamed IPs. The curves are plotted with a logarithmic scale on the y-axis to emphasize the tail of the distribution. The inflation ratio on the x-axis starts from 2 because we want to focus on the paths that are severely inflated. We can clearly see that a small number of misnamings introduce many unusually long paths. For the paths with inflation ratio over four, more than 50% of them are miscalculated due to misnamed interfaces. We also examined the length of these severely inflated paths. Among the paths with inflation ratio over 2, roughly 60% of them have a direct distance longer than 500 miles. This means their inflated distance is longer than 1000 miles.

6 Related Work

The pioneering work of Rocketfuel provides techniques for inferring detailed ISP topology using traceroutes [12]. In their work, they tried to filter out false edges by removing the links whose distance to latency ratio exceeds the speed of light. Although this heuristic helps to remove certain false edges, it may still miss those less obvious ones due to the reasons we discussed in Section 5.2. In a later work, Teixeira *et al.* found that the Rocketfuel topology of Sprint has significantly higher path diversity than the real topology because of extra false edges [14]. Since path diversity directly impacts the resilience of a network to failures, such over-estimated path diversity may severely mislead network

designers and operators. They suspected this is due to imperfect alias resolution. However, this still cannot explain the POP-level false edges. Our work complements these existing works by identifying that DNS misnamings could be a major source of POP-level false edges. We also propose ways to fix the misnamings.

7 Conclusion & Discussion

We have shown that DNS misnaming, a relatively harmless problem from the network operator's standpoint, can be a much more serious problem for network researchers. A small fraction of misnamed router interfaces gets magnified, leading to a larger fraction of false links in the inferred connectivity graph. These links then cause errors in the path inflation metrics, leading to a mistaken belief that the routing decisions are worse than they actually are. The approaches we have developed to identify and correct the misnaming are able to resolve the problems that have significant impact on topology mapping and path inflation and we have verified them in consultation with a major ISP. Our future plans include conducting similar study on other major ISPs, and to expand the scope of the problems examined.

One of the other inferred metrics that is likely to be affected by these misnamings is path asymmetry [6]. Even if packets traverse the exact same set of links in both directions, the addresses reported by traceroute will differ in the two directions, so a misnaming of a single interface will give the appearance of asymmetric paths. While we are interested in determining how much false asymmetry arises from misnamed interfaces, it requires cooperation at both endpoints to generate and compare traceroute traffic in both directions. Our current infrastructure does not provide this capability, since we do not control the destination endpoint. It may be possible to model a large ISP and use intra-AS routing information to separate the causes of perceived asymmetry, but this effort requires more explicit data from the ISP than we currently have. Our current approach only uses explicit information from the ISP for verification, not for problem identification.

Additionally, misnaming may provide a false sense of security when inferring shared fate of links—mislabeled may give someone the mistaken impression that two paths with the same source and destination traverse different cities, and would therefore not use the same physical POPs. Especially in the cases where real links exist between the cities, even a moderately careful inspection would provide a false impression that the paths do not share fate. In this scenario, misnaming could affect an organization's disaster recovery planning, rather than affecting the analyses of external researchers.

Our larger goal is to raise awareness of this kind of problem so that network researchers performing

inference-based analysis become aware of the possibility that a large number of anomalous results may stem from a small number of input errors, instead of automatically assuming that the network itself is anomalous. Beyond just prodding other researchers to re-examine their approach in using DNS names for topological or geographic data, our longer-term goals are to stimulate new research into automatically detecting and resolving these problems, as well as to identify other research areas where this kind of mislabeling may exist. Given how easily unchecked DNS errors can cause serious misinterpretations of traceroute data, we believe that other network measurement may be similarly affected.

8 Acknowledgments

We would like to thank Nick Feamster for his comments on an earlier draft of this paper, and our shepherd Geoff Voelker and the anonymous reviewers for their useful feedback. This work was supported in part by NSF Grants ANI-0335214, CNS-0439842, and CNS-0520053.

References

- [1] P. Barford and W. Byrd. Interdomain routing dynamics. *Unpublished report*, June 2001.
- [2] N. Feamster, D. G. Andersen, H. Balakrishnan, and M. F. Kaashoek. Measuring the effects of Internet path faults on reactive routing. In *Proc. ACM SIGMETRICS*, June 2003.
- [3] Z. Mao, J. Rexford, J. Wang, and R. H. Katz. Towards an accurate AS-level traceroute tool. In *Proc. ACM SIGCOMM*, 2003.
- [4] V. N. Padmanabhan and L. Subramanian. An investigation of geographic mapping techniques for Internet hosts. In *Proc. ACM SIGCOMM*, Aug. 2001.
- [5] U. of Oregon RouteViews Project. <http://www.routeviews.org>.
- [6] V. Paxson. End-to-end routing behavior in the Internet. In *Proc. ACM SIGCOMM*, Aug. 1996.
- [7] PlanetLab. <http://www.planet-lab.org>.
- [8] RIPE. <http://www.ripe.net>.
- [9] RouteServer. <http://www.bgp4.net/>.
- [10] ScriptRoute. <http://www.scriptroute.org/>.
- [11] N. Spring, R. Mahajan, and T. Anderson. Quantifying the Causes of Path Inflation. In *ACM SIGCOMM*, Aug. 2003.
- [12] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *Proc. ACM SIGCOMM*, Aug. 2002.
- [13] L. Subramanian, V. N. Padmanabhan, and R. H. Katz. Geographic Properties of Internet Routing. In *Proc. USENIX Annual Technical Conference*, June 2002.
- [14] R. Teixeira, K. Marzullo, S. Savage, and G. Voelker. In Search of Path Diversity in ISP Networks. In *Proc. Internet Measurement Workshop*, Oct. 2003.
- [15] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. Wu, and L. Zhang. An analysis of BGP multiple origin AS (MOAS) conflicts. In *Proc. Internet Measurement Workshop*, Nov. 2001.

Efficient Query Subscription Processing for Prospective Search Engines

Utku Irmak *

Svilen Mihaylov †

Torsten Suel *

Samrat Ganguly ‡

Rauf Izmailov ‡

Abstract

Current web search engines are retrospective in that they limit users to searches against already existing pages. Prospective search engines, on the other hand, allow users to upload queries that will be applied to newly discovered pages in the future. Some examples of prospective search are the subscription features in Google News and in RSS-based blog search engines.

In this paper, we study the problem of efficiently processing large numbers of keyword query subscriptions against a stream of newly discovered documents, and propose several query processing optimizations for prospective search. Our experimental evaluation shows that these techniques can improve the throughput of a well known algorithm by more than a factor of 20, and allow matching hundreds or thousands of incoming documents per second against millions of subscription queries per node.

1 Introduction

The growth of the world-wide web to a size of billions of pages, and the emergence of large-scale search engines that allow real-time keyword searches over a large fraction of these pages, has fundamentally changed the manner in which we locate and access information. Such search engines work by downloading pages from the web and then building a full-text index on the pages. Thus, they are *retrospective* in nature, as they allow us only to search for currently already existing pages – including many outdated pages. In contrast, *prospective search* allows a user to upload a query that will then be evaluated by the search engine against documents encountered in the future. In essence, the user subscribes to the results of the query. The user can be notified of new matches in one of several ways, e.g., via email or an RSS reader.

A naive implementation of a prospective search engine

might simply execute all the subscription queries periodically against any newly arrived documents. However, if the number of subscriptions is very large, this would result either in a significant delay in identifying new matches, if we only execute the queries very rarely, or a significant query processing load for the engine. Following the approach in [11], we essentially reverse the roles of the documents and the queries. That is, we build an inverted index on the subscriptions instead of the documents, and then issue a number of queries into the index for each newly arriving document. We note, however, that the two cases are not completely symmetric. In this paper, we study techniques for optimizing the performance of prospective search engines. A more detailed version is available from the first author.

Applications of Prospective Search: One of the popular implementations of prospective search is the *News Alert* feature in Google News. It allows users to subscribe to a keyword search, in which case they will be notified via email of any newly discovered results matching all the terms. Similar services include specialized search applications created for job or real estate searches. Prospective search can be performed with the help of RSS (RSS 2.0: Really Simple Syndication) feeds, which allow web sites to syndicate their new content at a specified URL. Thus, a prospective search engine can find the new content on a site by periodically downloading the appropriate RSS feed. There are a number of existing weblog and RSS search engines based on RSS feeds, including PubSub, Bloglines, Technorati, and Feedster.

Problem Setup: We are given n queries q_0 to q_{n-1} , where each query q_i contains s_i terms (keywords) $t_{i,0}, \dots, t_{i,s_i-1}$. We define T as the union of all the q_i , i.e., the set of terms that occur in at least one query. The terms in a query may be arranged in some Boolean formula, though we will mainly focus on the AND queries. Given these queries, we are allowed to precompute appropriate data structures, say, an inverted index.

After preprocessing the queries, we are presented with a sequence of documents d_0, d_1, \dots , where each document d_j is a set of terms. We assume that $d_j \subseteq T$ for all j ; this can be enforced by pruning from d_j any terms that do not occur in any query q_i . We process the documents one by one where for each d_j we have to determine all q_i such that d_j matches q_i . Within our matching system,

*CIS Department, Polytechnic University, Brooklyn, NY 11201. {uirmak@cis.poly.edu, suel@poly.edu}. The third author was also partially supported by NSF Awards IDM-0205647 and CCR-0093400, and the New York State Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University.

†CIS Department, University of Pennsylvania, Philadelphia, PA 19104. svilen@seas.upenn.edu.

‡NEC Laboratories America, Inc., Princeton, NJ 08540. {samrat@nec-labs.com, rauf@nec-labs.com}.

queries are assigned integer query IDs (QID), and documents are assigned integer document IDs (DID), and all terms in the queries and documents are replaced by integer term IDs (TID). The output of the matching process is a stream of (QID, DID) pairs indicating matches.

While retrospective search engines typically store their index on disk, we assume that for prospective search, all the index structures fit into main memory. Tens of millions of queries can be kept in memory on a single machine with the currently available memory sizes. In case of more queries, our results indicate that CPU cycles become a bottleneck before main memory.

Discussion of Query Semantics: Most current search engines assume AND semantics, where a query matches any document containing all query terms, in combination with ranking. As we show, AND queries can be executed very efficiently in an optimized system for prospective search; moreover, several other interesting types of queries can be efficiently reduced to AND queries. Therefore, we focus on AND queries.

Boolean queries involve a number of terms connected by AND, OR, and NOT operators. In our framework, they are executed by converting them to DNF, inserting each conjunction as a separate AND query, and then removing duplicate matches from the output stream.

In the RSS search scenario, it might be preferable to restrict the keyword queries to certain fields. It is well understood that inverted index structures with appropriate extensions can efficiently process many such queries for retrospective search, and similar extensions are also possible for prospective search.

Contributions of this Paper:

- We describe the design of a high-performance subscription matching processor based on an inverted index, with several optimizations based on term frequencies and a Bloom filter structure.
- We study preprocessing schemes that cluster subscriptions into *superqueries*.
- We evaluate our schemes against a search engine trace, and detail the performance improvements created by our optimizations.

2 The Core Query Processor

In our query processor, TIDs are assigned from 0 to $|T| - 1$, and the terms in each query are ordered by TID; thus we can refer to the first, second, etc., term in a query. Any incoming documents have already been preprocessed by parsing out all terms, translating them into TIDs, and discarding any duplicate terms or terms that do not occur in any query. It will also be convenient to assume that QIDs are assigned at random. Note that

we expect additions and deletions of subscriptions to be handled in an amortized fashion, by periodic rebuilding of the structures (including updating the assignments of TIDs and QIDs).

The main data structure used in all our algorithms is an *inverted index*, which is also used by retrospective search engines. However, in our case we index the queries rather than the documents, as proposed in [11]. The inverted index consists of $|T|$ inverted lists, one for each unique term that occurs in the queries. Each list contains one posting for each query in which the corresponding word occurs, where a posting consists of the QID and the position of the term in the query (recall that terms are ordered within each query by TID). The QID and position can usually be stored together in a single 32-bit integer, and thus each inverted list is a simple integer array.

2.1 A Primitive Matching Algorithm

We now describe the primitive matching algorithm, which (with some variations) has been studied in a number of previous works including [8, 11, 9, 7]. The basic idea is as follows. We initially build the inverted index from the queries using standard index construction algorithms; see, e.g., [10]. We also reserve space for a hash table, indexed by QIDs, of some sufficient size. Given an incoming document consisting of a set of terms, we first clear the hash table, and then process the terms in the document in some sequential order. To process a term, we traverse the corresponding inverted list in the index. For each posting of the form (QID, position) in this list, we check if there is already an entry in the hash table, with an associated accumulator (counter) set to 1. If an entry already exists, we increase its accumulator by 1. This first phase is called the *matching phase*.

In the second phase (*testing phase*), we iterate over all created hash table entries. For every entry, we test if the final value of the accumulator is equal to the number of query terms; if so then we output the match between this query and the document. Note that for Boolean queries other than AND, we could reserve one bit in the accumulator for each term in the query, and then instead of increasing a counter we set the corresponding bit; in the testing phase we check if the accumulator matches the query through tests applying appropriate bit masks. Also, since QIDs are assigned at random, we can use the QID itself as our hash function for efficiency.

2.2 Optimizations for Primitive Algorithm

Exploiting Position Information and Term Frequencies: One problem with the primitive algorithm is that it

creates an entry in the hash table for any query that contains at least one of the terms. This results in a larger hash table that in turn slows down the algorithm, due to additional work that is performed but also due to resulting cache misses during hash lookups.

To decrease the size of the hash table, we first exploit the fact that we are focusing on AND queries. Recall that each index posting contains (QID, position) pairs. Thus, if we are processing a posting with a non-zero position, then this means that the term is not the term with the lowest TID in the query. Suppose we process the terms in the incoming document in sorted order, from lowest to highest TID. This means that for a posting with non-zero position, either there already exists a hash entry for the query, or the document does not contain any of the query terms with lower TID, and thus the query does not match. So we create a hash entry whenever the position in the posting is zero, and only update existing hash entries otherwise. As we will see, this results in significantly smaller hash table sizes. A further reduction is achieved by simply assigning TIDs to terms in order of frequency, that is, we assign TID 0 to the term that occurs the least frequent in the set of queries, and TID $|T| - 1$ to the most frequent term. This means that an accumulator is only created for those queries where the incoming document contains the least frequent term in the query.

To implement this efficiently, we split each inverted list into two parts, a smaller list containing only postings with positions equal to zero, and the rest of the list. We then perform two passes over the terms in the incoming document, the first pass generates the hash entries, and the second pass updates the existing entries. This simplifies the critical inner loop over the postings and also allows us to quickly determine the optimal hash table size for each incoming document, by summing up the lengths of the first parts of the inverted lists involved.

Bloom Filters: As a result of the previous set of optimizations, hash entries are only created initially, and most of the time is spent afterwards on lookups to check for existing entries. Moreover, most of these checks are negative, i.e., the corresponding entry does not exist. In order to speed up these checks, we propose to use a Bloom filter [2, 3], which is a probabilistic space-efficient method for testing set membership.

We use a Bloom filter in addition to the hash table. In the matching phase, when hash entries are created, we also set the corresponding bits in the Bloom filter; the overhead for this is fairly low. In the testing phase, we first perform a lookup into the Bloom filter to see if there might be a hash entry for the current QID. If the answer is negative, we immediately continue with the next posting; otherwise, we perform a lookup into the hash table.

Use of a Bloom filter has two advantages. The Bloom

filter structure is small and thus gives better cache behavior, and the innermost loop of our matching algorithm is also further simplified. We experimented with different settings for the size of the Bloom filter and the number of hash functions; our results indicate that a single hash function (trivial Bloom filter) performs best.

Partitioning the Queries: We note that the hash table and Bloom filter sizes increase linearly with the number of query subscriptions, and thus eventually grow beyond the L1 or L2 cache sizes. This leads to our next optimization. Instead of creating a single index, we partition the queries into a number p of subsets and build an index on each subset. In other words, we partition the index into p smaller indexes. An incoming document is then processed by performing the matching sequentially with each of the index partitions. While this does not decrease the number of postings traversed, or the locality for index accesses, it means that the hash table and Bloom filter sizes that we need are decreased by a factor of p .

2.3 Experimental Evaluation

Since we were unable to find any large publicly available query subscription logs, we decided to use Excite search engine query logs, collected in 1999. We note that query subscriptions in a prospective engine would likely be different in certain ways from standard retrospective queries; in particular, we would not expect as many extremely broad queries. For this reason, we will also look at how performance changes with query selectivity, by looking at different subsets of the query logs. To be used as incoming documents, we selected 10,000 web pages at random from a large crawl of over 120 million pages from Fall 2001.

We removed stop words and duplicate queries from the query trace, and also converted all the terms to lower case. We also removed queries that contained more than 32 terms; there were only 43 such queries out of a total of 1,077,958 distinct queries. Some statistics on the query logs, documents, and resulting inverted index lookups is as follows: There are 271,167 unique terms in the query log, and each query contains about 3.4 terms on average. The number of postings in the index is 3,633,970. Each incoming document contains about 144 distinct terms that also occur in the queries. For each document, our algorithms will visit about 200,000 postings, or about 1,400 postings per inverted list that is traversed. Of those postings, only about 6,630 have position zero if we assign TIDs according to term frequencies.

To experiment with numbers of queries beyond the size of the query log, we replicated the queries several times according to a *multiplier* between 1 and 14, for a maximum size of about 15 million queries. We note that

the core query processor does not exploit similarities between different queries, and thus we believe that this scaling approach is justified. Later, in Section 3, we will incorporate clustering techniques into our system that exploit such similarities; for this reason we will not use a multiplier in the later evaluation of the clustering schemes, which limits us to smaller input sizes.

In the experiments, we report the running times of the various optimizations when matching different numbers of queries against 10,000 incoming documents. The experiments are performed on a machine with a 3.0 Ghz Pentium4 processor with 16 KB L1 and 2 MB L2 cache, under a Linux 2.6.12-Gentoo-r10 environment. We used the gcc compiler with Pentium4 optimizations. We also used the vtune performance tools to analyze program behavior such as cache hit ratio etc. In the charts, we separately show the times spent on the matching and testing phases. Note that the matching phase includes all inverted index traversals and the creation and maintenance of the accumulators, while the testing phase merely iterates over the created accumulators to identify matches.

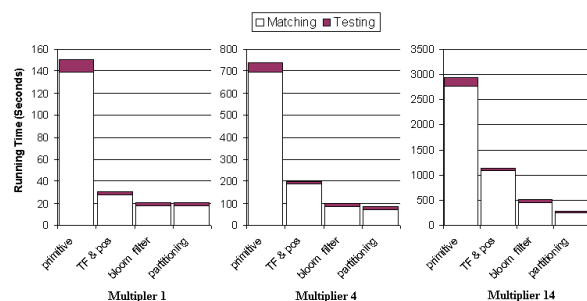


Figure 2.1: Running times of the various algorithm optimizations for different numbers of queries.

In Figure 2.1, we show the time spent by four versions of our query processor: (i) the primitive one, (ii) with optimization for AND and assignment of TIDs by frequency, (iii) with Bloom filter, and (iv) with index partitioning with optimal choice of the number of partitions. We show total running times in seconds for matching 10,000 documents against the queries with multipliers of 1, 4, and 14. At first glance, running times are roughly linear in the number of queries. More exactly, they are slightly more than linear for the first three algorithms, due to the increased sizes of the hash table and Bloom filter structures resulting in more cache misses, while the best algorithm (iv) remains linear by increasing the number of partitions.

As discussed, many Excite queries may have much larger result sizes than typical subscription queries would have. To examine possible relationships between matching performance and query selectivities, we partitioned our queries into quintiles according to selectivity. To do so, we matched all queries against a larger collection

of around 144,000 documents (disjoint from the set of 10,000 we used for testing), and counted the number of matches of each query. We then partitioned queries into five subsets, from the 20% with the fewest number of matches to the 20% with the most. In the Figure 2.2, we show how the running times of the algorithms change as we go from queries with very few results (leftmost 4 bars) to queries with very many results (rightmost 4 bars). Not surprisingly, queries with many matches are more costly. (Since we use a multiplier of 1, the partitioning does not seem to give any benefits in the figure.)

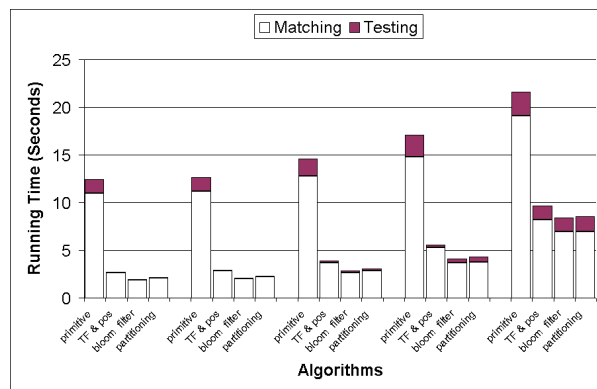


Figure 2.2: Running times versus query selectivities for the various algorithms, with multiplier 1.

To illustrate the benefits of index partitioning, we performed additional runs on a machine with 512 KB instead of 2 MB of L2 cache. As shown in Figure 2.3, index partitioning resulted in a gain by about a factor of 4 for the largest query set. On the 2 MB machine, a similar effect is expected for larger query multipliers.

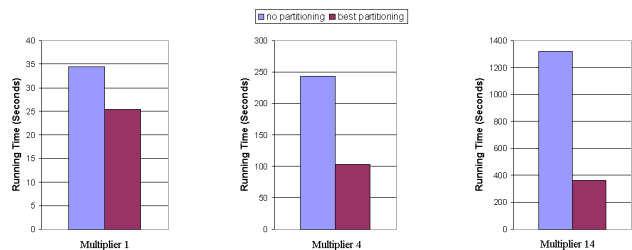


Figure 2.3: Benefit of best possible index partitioning on a machine with smaller L2 cache.

3 Optimizations using Query Clustering

In this section, we study clustering techniques to obtain additional performance benefits. We note that clustering of subscriptions has been previously studied, e.g., in [8], but in the context of more structured queries. Our simple approach for clustering subscriptions is as follows. In a preprocessing step, we cluster all queries into artificial *superqueries* of up to a certain size, such that every

query is contained in a superquery and shares the same least frequent term with a superquery (and thus with all other queries in the cluster). Then we present these superqueries to the query processor, which indexes them and performs matching exactly as before. Thus the resulting improvements are orthogonal to the earlier techniques. The only changes to the core subscription processor are as follows: (i) During matching, we set appropriate bits in the accumulators for each update instead of counting, and (ii) in the testing phase we need to test each query that is contained in a superquery that has an accumulator. To do this test, we create a simple structure for each superquery during preprocessing that contains a list of the QIDs of the contained queries, plus for each QID a bit mask that can be used to test the superquery accumulator for this subquery.

We now discuss briefly how clustering impacts the cost of matching. Consider any two queries (or superqueries) that share the same least frequent term. Note that by combining these two queries into one superquery, we are guaranteed to decrease the size of the index by at least one posting. Moreover, the number of hash entries created and accumulator updates performed during the matching phase will never increase but may often decrease as a result of combining the two queries. On the other hand, we need at least one bit per term in the superquery for the accumulator, and a very large superquery would result in higher costs for both testing and hash table accesses. However, this effect is not easy to capture with a formal cost model for clustering. Instead, we will impose an upper bound b on the size of each superquery, and try to minimize index size under this constraint. In general, this problem is still intractable, and we will focus on some heuristics.

3.1 Greedy Algorithms for Clustering

All the algorithms we present in this subsection follow the clustering approach discussed above. They start out by initially grouping all queries into pools based on their least frequent terms, and then separately build superqueries within each pool. Note that if we use index partitioning, we should make sure to assign all queries in a pool to the same partition.

Random Selection: The simplest approach starts with the empty superquery and then repeatedly picks an arbitrary query from the pool and merges it into the superquery, as long as the resulting superquery has at most $b = 32$ terms (apart from the least frequent term). If the result has more than b terms, then we write out the old superquery and start a new one.

Alphabetical Ordering: We first sort all queries in the pool in reverse alphabetical order - recall that the terms

in each query are sorted by frequency and thus we sort according to most common, second most common, etc. terms. We then consider queries in sorted order as we construct superqueries, rather than in arbitrary order.

Overlap Ratio: The third greedy algorithm builds superqueries by checking all remaining queries in the pool to find the one with the best match, i.e., a query that has a lot of terms in common with the current superquery.

3.2 Experimental Evaluation

We cannot use the multiplier method of Section 2.3 in the evaluation of the proposed clustering algorithms since they exploit the similarities among the queries in the collection. In order to obtain at least a slightly larger collection of queries, we combined the Excite queries with another set of queries from the AltaVista search engine. The combined set contains 3,069,916 queries and 922,699 unique terms. The number of postings in the resulting index is 9,239,690. We first compare the performance of the proposed clustering algorithms in Table 3.1 when b is set to 32 terms. The best algorithm from the previous section is shown on the last row. Even the most naive clustering algorithm gives significant benefits, and the algorithm based on overlap outperforms all others.

	Matching	Testing	Total
Random	16.83	1.35	18.18
Alphabetical	14.42	1.34	15.76
Overlap	13.71	1.34	15.05
Best non-clustered	33.28	6.34	39.62

Table 3.1: Running time of the clustering algorithms on the combined query set (in seconds).

In Table 3.2, we show the number of superqueries created by each clustering algorithm, the number of postings in the resulting inverting index, and the number of accumulators created during the matching process. As we see, clustering decreases the number of index postings by about 40%, but the number of accumulators created during the matches is reduced by a factor of up to 20. This is because clustering is most effective for large query pools that have a significant number of queries with the same least common query term that can be combined.

	(Super) queries	Postings	Accumulators
Random	957366	6089165	8870966
Alphabetical	948417	5784684	7670446
Overlap	939779	5522510	6543883
Best	3069916	9239690	130691462

Table 3.2: Comparison of the clustering algorithms and the best algorithm.

Next, we investigated if there are benefits in allowing

larger superqueries with up to 64, 96, and 128 terms. We observed that there is a slight benefit in allowing up to 64 terms, but for 96 and 128 terms any further gains are lost to additional testing time. We also tried to improve the greedy approach based on overlap by looking further ahead at the next 2 and 3 sets that can be added, as opposed to a strictly one step at a time approach. However, we did not observe any measurable gains, indicating that maybe the overlap approach is already close to optimal. We ran the clustering algorithms on the different selectivity ranges, introduced in Section 2.3, which showed that again queries with many results are more expensive to match (we omit the figures due to space limitations). Finally, we summarize our results by showing the throughput rates in documents per second obtained by our various matching algorithms in Figure 3.1, on the Excite set and the combined set of Excite and AltaVista queries. We see that overall, throughput increases by more than a factor of 20 when all techniques are combined.

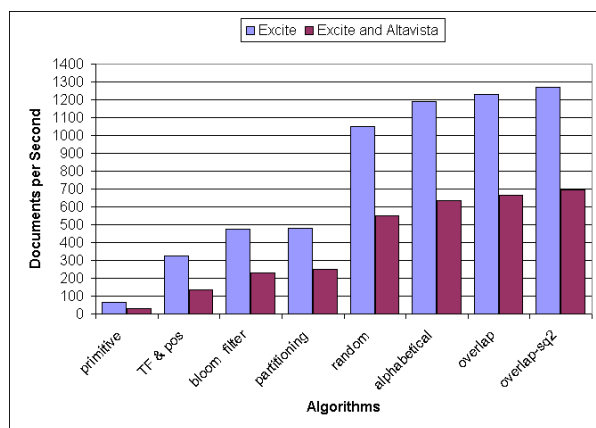


Figure 3.1: Number of documents processed per second for Excite and for the combined query set. (In overlap-sq2, the superqueries can have up to 64 terms, using two unsigned integers.)

4 Related Work

Our work is most closely related to the SIFT project in [11], which also focuses on keyword queries and uses an inverted index structure on the queries. In SIFT, the emphasis is on ranked queries and the queries are represented in the vector space model using OR semantics. Thus, users can specify term weights and a relevance threshold (e.g., cosine measure) that are then used in identifying matching documents.

A main memory algorithm for matching events against subscriptions is proposed in [8], where an event is an attribute/value pair, and a subscription is a conjunction of (attribute, comparison operator, constant) predicates.

The proposed algorithm also employs a clustering approach. The created clusters have access predicates, where an access predicate is defined as a conjunction of equality predicates. In our approach, we create clusters with new artificial superqueries. The scenario we consider is different, as the terms in the queries as well as the content of the incoming documents are keywords.

Another body of related work is in the area of content-based networking and publish/subscribe communication systems [1, 6]. In this model, subscribers specify their interests by conjunctive predicates, while sources publish their messages as a set of attribute/value pairs. The goal is to efficiently identify and route messages to the interested subscribers [5, 4]. The forwarding algorithms used by the routing nodes are related to our query processing algorithm; see [7]. Previous related work exists in the database literature about triggers and continuous queries; in stream processing and XML filtering systems.

5 Acknowledgments

We thank the anonymous Usenix reviewers and our shepherd John Reumann for their comments and insights.

References

- [1] R. Baldoni, M. Contenti, and A. Virgillito. The evolution of publish/subscribe communication systems. In *Future Directions of Distributed Computing*, volume 2584 of *LNCS*. Springer, 2003.
- [2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [3] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Proc. of the 40th Annual Allerton Conf. on Communication, Control, and Computing*, pages 636–646, 2002.
- [4] F. Cao and J. P. Singh. Efficient event routing in content-based publish-subscribe service networks. In *Proc. of IEEE Infocom Conf.*, 2004.
- [5] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *Proc. of IEEE Infocom Conf.*, 2004.
- [6] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, 2001.
- [7] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Proc. of ACM Sigcomm*, 2003.
- [8] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. of ACM Sigmod Conf.*, 2001.
- [9] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *Conf. on Cooperative Information Systems*, 2000.
- [10] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.
- [11] T. W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM Transactions on Database Systems*, 24(4):529–565, 1999.

IP Only Server

Muli Ben-Yehuda¹ Oleg Goldshmidt¹ Elliot K. Kolodner¹ Zorik Machulsky¹
Vadim Makhervaks² Julian Satran¹ Marc Segal³ Leah Shalev¹
Ilan Shimony¹

IBM Haifa Research Laboratory

¹{mulib, olegg, kolodner, machulsk, satran, leah, ishimony}@il.ibm.com

²vadim.makhervaks@gmail.com

³marcs@cs.technion.ac.il

Abstract

Present day servers must support a variety of legacy I/O devices and protocols that are rarely used in the day to day server operation, at a significant cost in board layout complexity, reliability, power consumption, heat dissipation, and ease of management. We present a design of an IP Only Server, which has a single, unified I/O interface: IP network. All of the server's I/O is emulated and redirected over IP/Ethernet to a remote management station, except for the hard disks which are accessed via iSCSI. The emulation is done in hardware, and is available from power-on to shutdown, including the pre-OS and post-OS (crash) stages, unlike alternative solutions such as VNC that can only function when the OS is operational. The server's software stack — the BIOS, the OS, and applications — will run without any modifications.

We have developed a prototype IP Only Server, based on a COTS FPGA running our embedded I/O emulation firmware. The remote station is a commodity PC running a VNC client for video, keyboard and mouse. Initial performance evaluations with unmodified BIOS and Windows and Linux operating systems indicate negligible network overhead and acceptable user experience. This prototype is the first attempt to create a diskless and headless x86 server that runs unmodified industry standard software (BIOS, OS, and applications).

1 Introduction

Present day server systems support the same set of I/O devices, controllers, and protocols as desktop computers, including keyboard, mouse, video, IDE and/or SCSI hard disks, floppy, CD-ROM, USB, serial and parallel ports, and quite a few others. Most of these devices are not utilized during the normal server operation. The data disks are frequently remote, and both Fibre Channel and iSCSI now support booting off remote disk devices, so directly

attached hard disks are not necessary for operating system (OS) boot either. Removable media devices, such as floppies and CD-ROMs, are only used for installation of OS and applications, and that can also be avoided with modern remote storage management systems.

Moreover, there are no users who work directly on the server, using keyboard, mouse, and display — normal administrative tasks are usually performed over remote connections, at least while the server is operational. Remote management is done via protocols such as Secure Shell (SSH) and X for Linux/UNIX systems, Microsoft's Windows Terminal Services, and cross-platform protocols such as the Remote Framebuffer (RFB, see [6]), used by the popular Virtual Network Computing (VNC) remote display scheme. However, local console access is still required for some operations, including low-level BIOS configuration (pre-OS environment) and dealing with failures, such as the Windows "blue screen of death" and Linux kernel panics (post-OS environments). Local console is usually provided either via the regular KVM (keyboard-video-mouse) interface or a serial line connection.

The legacy protocols and the associated hardware have non-negligible costs. The board must contain and support the multitude of controllers and the associated auxiliary electrical components each of them requires. This occupies a significant portion of the board real estate that could otherwise contain, say, an additional CPU or memory. The multitude and complexity of the legacy components also reduce the mean time between failures (MTBF).

We propose that future servers will only need CPUs, memory, a northbridge, and network interface cards (NICs). All the legacy I/O that is done today, e.g., over PCI, will be done over a single, universal I/O link — the ubiquitous IP network. All communication with storage devices, including boot, will be done over iSCSI, console access will also be performed over the network. Protocols such as USB can also be emulated over IP [5], pro-

viding a variety of remote peripherals such as CD-ROM, printers, or floppy if they are needed.

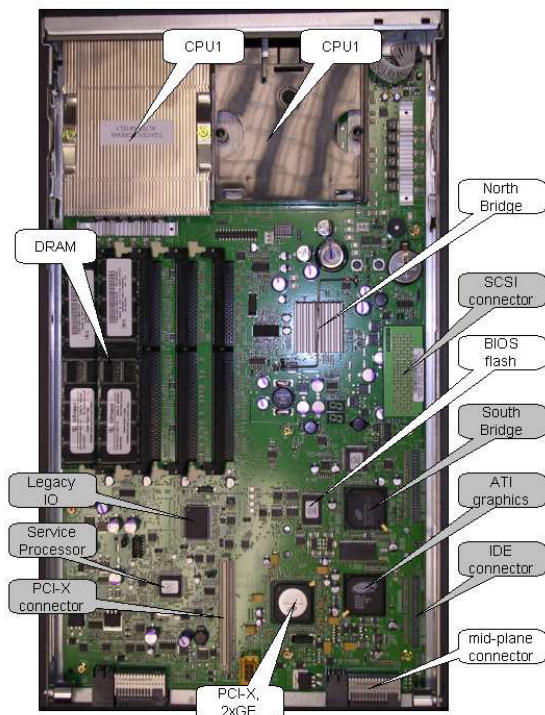


Figure 1: Components of an IBM HS20 blade server.

To illustrate this point, Figure 1 identifies the various components of an x86 server (an IBM HS20 blade server chosen for its unobstructed layout). The CPUs, the DRAM, the northbridge, the BIOS flash, and the network hardware are necessary, while the southbridge, the SCSI and IDE controllers, the graphics adapter, and the integrated legacy I/O chip (implementing the keyboard and mouse controllers, various timers, etc.) can be removed and their functionality can be emulated over the network.

This “remoting” of I/O can be achieved without any modifications at all to the applications, the OS, or the BIOS of the server if the protocol emulation is done in hardware. Substituting a single hardware component for all the legacy controllers, capturing all the bus transactions involving the legacy devices, remoting the transactions over the IP network, and performing the actual I/O at remote systems will be completely transparent to the BIOS and the OS, and thus to the applications.

While many software based alternatives exist for remoting I/O when the OS is up and running (see Section 2), doing the protocol emulation in hardware is essential for supporting the pre-OS (e.g., BIOS or boot-loader) and post-OS environments.

We developed a research prototype of such an “IP

Only Server” that uses IP for all of its I/O needs. We designed and implemented a legacy I/O emulator based on a COTS FPGA. The FPGA, connected to the host via the PCI bus, serves as the local keyboard, mouse, and VGA controller. All keyboard, mouse, and VGA traffic reaches the FPGA and is sent to a remote station over IP. The user is able to perform all the management operations — throughout the lifetime of the server, i.e., during boot, BIOS, OS initialization, normal operation, and post-OS stages — from the remote station. No changes were needed for the software running on the host. In particular, neither the BIOS nor the OS were modified.

The performance of the prototype is acceptable for the usual server management tasks. The only significant network load may come from the remote iSCSI storage, the network utilization due to remote management is small, and the user experience is acceptable.

2 Related Work

The concept of allowing the user to interact with a remote computer over a network had a long history. Today there exists a wide variety of thin clients, e.g., SSH/X11, VNC, SunRay, Citrix Metaframe, and Windows Terminal Server. Our approach differs from all these solutions in two major ways: we allow remoting of legacy I/O over the network a) without any host software modifications, and b) from the moment the computer has powered on until it has powered off, including when no operating system is present (BIOS stage as well as OS crash).

The Super Dense Server research prototype [7] presented servers with no legacy I/O support. It used Console Over Ethernet, which is OS dependent, supported text mode only, ran Linux, and used LinuxBIOS [4] rather than a conventional BIOS. In contrast, the IP Only Server runs unmodified OSes and BIOSes and supports graphical VGA modes as well as text based modes. Attempting sweeping changes in the BIOS (e.g., switching to LinuxBIOS) while requiring to support a wide variety of boards and many different software stacks would adversely affect reliability, availability, serviceability, and system testing.

“USB over IP” [5] is used as a peripheral bus extension over an IP network. USB/IP requires a special OS-specific driver and thus is only available when the OS is operational, while the IP Only Server does not require OS modifications — it listens on the PCI bus for transactions using a hardware component.

Baratto et al. presented THINC [3], a remote display architecture that intercepts application display commands at the OS’s device driver interface. This approach looks promising for remoting the display while the OS is running, but does not handle either pre-OS or exception (post-OS) conditions. THINC could be used together

with modifications to the system's BIOS to build a pure software IP Only Server. However, by using specialized hardware the system can be remoted at all times.

IBM's JS20 PowerPC-based blades do not contain any video/mouse/keyboard components. Instead Serial-over-LAN (tunneling text-only serial console messages over UDP datagrams) is used. KVM over IP products (e.g., Cyclades AlterPath KVM/net) provide an easy way to remote all operating environments. However, such products carry a non-negligible price tag, and servers using KVM over IP still require a full complement of hardware components.

3 Design

The IP Only Server was designed with several guidelines in mind.

1. The server must run unmodified software, including OS and BIOS.
2. Remote access is needed at all times, from the BIOS boot stage through the OS's lifetime, and even post-OS environments such as the "blue screen of death" or a Linux kernel oops. This does not preclude a more effective remote access method when the OS is operational, such as X-Windows, or Windows Terminal Server.
3. The server should have the minimal amount of local state required for disconnected operation. The hard drives should be remoted over IP, including boot.
4. The IP Only Server must be able to work even when no remote management station is connected, or when one has been connected and then disconnected. Obviously, the remote storage that is necessary for boot and normal operation of the server must be available at all times.
5. Text (console) and graphical mode support must be provided. There is no requirement to provide more than plain VGA mode support — the IP Only Server is not aimed at users who need accelerated graphics.
6. The remote management station should not require a custom or proprietary client, e.g., the KVM-over-IP (Keyboard/Video/Mouse-over-IP) protocol should be based on open standards.
7. A single remote station should be able to control multiple IP Only Servers concurrently.

The IP Only Server can be based on any standard architecture (such as x86). The CPU, memory, north-bridge, and BIOS flash are not modified. The server

will include at least one network interface. Other peripheral components will not be needed. The functionality of the peripheral components can be emulated by dedicated logic that presents the legacy I/O interfaces to the host (via a PCI bus), and remotes them over an IP based protocol to the remote station. The logic may be implemented as an ASIC or an FPGA depending on the cost/programmability trade-off.

The IP Only Server will not include any local disks. Instead, it boots from a remote boot device, such as an iSCSI disk via iBOOT [1] or PXE. Alternatively, disk access can be remoted like the other legacy I/O protocols. A mixture of the two approaches is possible in principle: the emulation hardware can include an implementation of a boot-capable iSCSI initiator. This will leave the BIOS flash as the only local state.

For the prototype described below we designed an FPGA that presented itself as a VGA/keyboard/mouse device. The server's BIOS and OS accessed the FPGA using their standard drivers. The FPGA received all host accesses as PCI transactions, and handled them appropriately.

We experimented with two different approaches to remoting these PCI transactions to a remote station. The first approach, the Internet PCI Remote Protocol (iPRP) was essentially "PCI over IP": PCI transactions were wrapped by IP packets, sent to the remote station, and processed there. Responses were sent back as IP packets as well and passed them to the local PCI bus. Clearly iPRP does not satisfy design guideline 4 above, and was used mainly as an intermediate debugging tool. It is described in Section 4.1.

The second approach, using the RFB protocol, is described in Section 4.2. In this scheme the emulation FPGA translates keyboard, mouse, and video PCI transactions into the high level RFB protocol that allows using any VNC client (and any OS) on the remote station. PCI transactions are processed locally by the FPGA, while the display and user inputs are handled by the remote station.

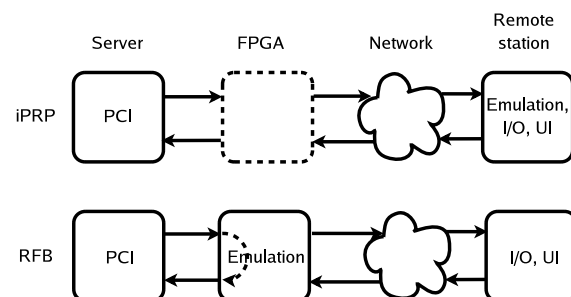


Figure 2: Comparison of iPRP and RFB implementations of an IP Only Server.

The difference between the two schemes is highlighted by Figure 2: with iPRP the FPGA is essentially transparent, and the emulation is done in the remote station; with RFB the remote station exists for the user interface only, the emulation is done in the FPGA, and some PCI transactions (reads) are handled locally inside the FPGA.

Thus, in the RFB design, if the user is not interested in interacting with the server, the server can operate without a remote station. On the other hand, a user can open VNC sessions against multiple IP Only Servers simultaneously. Clearly, this approach supports design guidelines 4 through 7.

4 Implementation

For the prototype we used a Dini-group DN3000k10s FPGA evaluation board with a Metanetworks Inc. MP1000TX Ethernet prototyping plug-in board. The FPGA is a Xilinx Virtex-II x2v8000. The design ran at 50 MHz, used 378 KB of local memory, and 8400 logic slices (equivalent to about 1M gates).

The FPGA firmware was divided into four main modules: a PCI Interface Module, a BIOS expansion ROM, a Network Interface Module, and a Transaction Processing Module.

The PCI interface module answers to keyboard controller addresses, VGA controller, VGA memory, and implements an expansion ROM base address register.

The device identifies itself as a PCI-based VGA adapter. Upon discovery of the VGA add-in card, standard BIOSes consider it as the default VGA adapter, and configure the chipset in such a way that all VGA I/O and memory transactions are routed to the device. Having keyboard controller I/O addresses routed to the device is trickier, and requires additional northbridge and I/O bridge configuration done by the BIOS. The expansion ROM is size 32kB, and contains the VGA BIOS routines. It was based on the GPL VGA BIOS included in the Bochs x86 emulator (<http://bochs.sourceforge.net>).

The network interface consists of a Fast Ethernet MAC and a DMA engine. The amount of the management traffic is small, but it may be beneficial to use a separate port for other reasons — for instance, if the main interface is down or saturated (e.g., due to heavy load or to a denial of service attack). The iSCSI storage interface may also be separate for performance and/or security reasons.

In our prototype implementation we used the FPGA's Ethernet interface for KVM emulation, and the server's regular Ethernet interface for the other traffic, including iSCSI.

The Transaction Processing firmware consists of a single loop that gets PCI transactions from the PCI interface, and either handles them locally or wraps them into

the appropriate network protocol (iPRP or RFB — see Sections 4.1 and 4.2 below).

To improve network utilization subsequent write transactions are coalesced into bigger network packets, while read transactions are either handled locally (RFB version) or sent to the network immediately (iPRP version). In the iPRP version read responses are forwarded to the PCI Interface Module that handles the PCI read response. The iPRP version also supports remote keyboard generated interrupt requests (IRQ1). Since this interrupt is reserved for the local keyboard controller we used the chipset's Open HCI USB legacy keyboard emulation feature to emulate it.

4.1 iPRP — Internet PCI Remote Protocol

The iPRP protocol uses the UDP protocol to emulate memory and I/O space PCI transactions over an IP network. The protocol was designed for ease of implementation, initial debugging and bring up of the hardware and firmware. It is not very efficient nor robust (a network problem or a remote side failure will cause the host to crash or just lock up).

A *command* is defined as a single I/O command such as memory read, I/O space write, or acknowledgment. Multiple commands may be packed into a single Ethernet frame. A *message* is one or more commands, mapped into a single Ethernet PDU. In a multi-command message only the last command may be a read type command, since the PCI-based system can not proceed until the read data arrives back. Each command has an attached *sequence number* (SN).

The protocol uses the 'go-back-N' scheme, i.e., there may be up to N unacknowledged commands in flight. An ACK message acknowledges all commands with sequence numbers less than the ACK's SN; in case of a read command the ACK also contains the returned data. A message transmit is triggered by a read, a timeout, or in case the maximum message size is reached.

The remote station software was based on the Bochs open source x86 emulator. We extracted the relevant PCI device emulation code and fed it the PCI transactions received over the iPRP payload as input. For host PCI reads, a return packet with the response is sent back to the FPGA.

4.2 RFB — Remote Frame Buffer

To overcome the shortcomings of the iPRP version, especially the fact that the server cannot operate without the remote station, we have to emulate the device controllers in the FPGA firmware rather than in the remote station software. As in the iPRP version we based the implementation on code from the Bochs emulator.

To transfer the keyboard, mouse, and video events between the FPGA and the remote station we chose the Remote Framebuffer (RFB) protocol [6]. To this end, we implemented a VNC server in the FPGA firmware. We choose RFB because it is a well known and widely used open protocol with numerous open source clients.

Our FPGA platform was limited in both space (350 KB of memory total) and speed (50 MHz CPU, no memory caches). Accordingly, we started with a straightforward hardware VGA controller emulation in software, based on the Bochs VGA controller, and optimized it both in space and time to fit our FPGA environment. For instance, by removing support for VGA modes that were not used by either Linux or Windows, we managed to reduce the VGA framebuffer to 128 KB.

Since the RFB protocol is TCP/IP based, we also added a TCP/IP stack to the FPGA firmware. Due to the limited memory and processing resources of the FPGA we had to implement a custom embedded TCP/IP stack. The stack is minimalistic and is specifically designed to fit our firmware environment, but it implements a complete TCP state machine and is streamlined: it has a very low memory footprint and avoids copying of data as much as possible.

Like the iPRP version, the RFB-based FPGA firmware is based on a single looping execution thread. The firmware receives the host's PCI transactions from the PCI interface. Host PCI reads are answered immediately, while PCI writes update the local device state machines. Every so often, the frame buffer updates are sent to the remote station to be displayed. The decision of when to update the remote station is crucial for establishing reasonable performance with our constrained FPGA; we developed heuristics that performed fairly well (cf. Section 5).

5 Performance Evaluation and Analysis

The most important performance metric for evaluating the IP Only Server is user experience, which is notoriously hard to quantify. In order to approximate the user's experience, we performed several measurements.

All tests were performed on two identical IBM x235 servers including the PCI-based FPGA evaluation board with a 100 Mb/s Ethernet network interface. The remote station software ran on two R40 Thinkpad laptops with a 1.4 GHz Pentium M CPU and 512 MB RAM each. The servers booted either Windows 2003 Server or Red Hat Enterprise Server Linux 3.0. The iSCSI connection was provided through a separate network interface, and the FPGA was not involved in communication with the iSCSI target at all. The performance of iSCSI storage has been studied independently, and we were primarily interested in the performance of our FPGA-based KVM

emulation. Therefore, for the purpose of these measurements we used local disks, to achieve a clean experimental environment.

First, we measured the wall time of a server boot for each of the three scenarios: a server with native legacy I/O peripherals, an IP Only Server with an FPGA using iPRP, and an IP Only Server with an FPGA using RFB. In each scenario, we measured the time from power on until a Linux console login prompt appeared, and from power on until a Windows 2003 Server "Welcome to Windows" dialog showed up.

As depicted in Table 1, an unmodified server booted to a Linux login prompt in 238 seconds while a server using the RFB version of the FPGA booted in 330 seconds. This is a 38% slowdown (for the RFB version), which is acceptable for the very first, unoptimized prototype. Additionally an unmodified server booted to the Windows 2003 Server "Welcome to Windows" dialog in 186 seconds and the server with the RFB FPGA booted in 289 seconds. This is slightly worse, a 55% slowdown, but again, it is acceptable.

Server	Native	iPRP	RFB
Linux	238	343 (144%)	330 (138%)
Windows	186	299 (160%)	289 (155%)

Table 1: Linux and Windows boot time in seconds.

It should be noted that the RFB and iPRP versions performed nearly the same. While iPRP sends much more traffic over the network than RFB the bulk of the emulation in the iPRP version is performed by the 1.4 GHz Pentium M CPU of the remote station, whereas in the RFB version it is done by the severely constrained 50 MHz FPGA CPU. This leads us to conclude that there exists a trade off between performance and cost here: by throwing a stronger (more expensive) FPGA or ASIC at the emulation, the performance can be easily improved.

We measured the network utilization using the RFB version of the FPGA in the the same Linux and Windows boot scenarios described above. Table 2 shows the number of actual data bytes exchanged, the number of TCP packets and the throughput for both Linux and Windows boot sequences. The throughput is of the order of 1 Mb/s — not a significant load for a modern local area network, and acceptable even for a wide area connection.

Booted OS	Unique Bytes	Packets	Throughput
Linux	52324982	189920	131488 B/s
Windows	47457592	155542	164384 B/s

Table 2: Network Utilization.

An additional observation is related to an important

class of systems that may be built on the basis of IP Only Servers—the so-called “hosted clients”. We ran some experiments to assess whether our low level display remoting scheme could be used to support a hosted client solution that deals with graphically intensive applications, and found, similarly to Baratto et al. [3], that remoting the higher level graphics access APIs was more effective than remoting the low level hardware graphic access. This indicates that there is even less reason to keep local graphics adapters on servers supporting hosted clients. For such servers our emulation scheme can be relegated to supporting the pre-OS environment and exception handling (possibly supporting more than one virtual console), i.e., functions that are not graphically intensive.

6 Future Work

Future IP Only Server work includes supporting more protocols, such as USB, serial, and parallel. USB support during pre-OS and post-OS stages is particularly desirable, as it would help provide support a diverse set of devices, including floppy and CD-ROM drives.

The IP Only Server prototype runs on x86 only at the moment. A port to other architectures such as PowerPC will validate the design.

The IP Only Server provides interesting opportunities when combined with virtualization technologies such as Xen [2] or VMWare. It can be used to give unmodified guests physical device access while providing requisite isolation and offering each guest its own view of the I/O hardware.

The IP Only Server could also provide a transparent virtualization layer on top of physical devices. It could provide several sets of device control registers associated with different virtual or physical machines. The host OS on each server would access what appears to it as a physical device, but is actually the IP Only hardware. The hardware or the remote station can direct the I/O to different real devices.

7 Conclusion

We present a novel approach to legacy I/O support in servers. We designed an IP Only Server, utilizing the IP network as the single I/O bus. All user interaction throughout the lifetime of the server, i.e., during boot, BIOS, OS initialization, normal operation, and post-OS (e.g., “blue screen of death”) stages are done from a remote station. No changes were needed for the software running on the host — in particular, neither the BIOS nor the OS were modified. While diskless servers have been attempted before, and headless servers exist as well (e.g.,

IBM’s JS20 blades), as far as we know this is the first attempt to create a diskless, headless server that runs industry standard firmware and software (BIOS, Windows or Linux OS) without any modifications.

We also found wider than expected dependencies on legacy devices, e.g., Windows boot touches keyboard and video hardware more than one would expect. Our emulation approach proved a good way to support Windows without requiring difficult and expensive (and not necessarily feasible) changes in the OS. One of the major reasons for the popularity of the x86 architecture is that it is affordable, and one reason for the low cost of x86 systems is ready availability of software built for large numbers of desktops. The emulation approach is instrumental in keeping this argument valid.

Our research prototype implementation of the IP Only Server provides a user experience that is comparable to that of a regular server, with reasonable latency and low network utilization.

The IP Only Server can provide significant savings in hardware and software costs, power consumption, heat dissipation and ease of management. It eliminates some legacy aspects of the PC architecture, replacing them with a single, simple, and modern counterpart.

Acknowledgments

The authors would like to thank Kevin Lawton and other Bochs developers for their excellent emulator and Michael Rodeh and Alain Azagury for their interest and support of this project.

References

- [1] iBOOT: Boot over iSCSI. <http://www.haifa.il.ibm.com/projects/storage/iboot/>.
- [2] B. DRAGOVIC, K. FRASER, S. HAND, T. HARRIS, A. HO, I. PRATT, A. WARFIELD, P. BARHAM, AND R. NEUGEBAUER. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), Bolton Landing, NY, pp. 164–177.
- [3] R. BARATTO, L. KIM, AND J. NIEH. THINC: a Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the 20th ACM Symposium on Operating systems* (2005), Brighton, United Kingdom, pp. 277–290.
- [4] R. MINNICH, J. HENDRICKS, AND D. WEBSTER. The Linux BIOS. In *Proceedings of the 4th Annual Linux Showcase and Conference* (2000), Atlanta, GA.
- [5] T. HIROFUCHI, E. KAWAI, K. FUJIKAWA, AND H. SUNAHARA. USB/IP — A Peripheral Bus Extension for Device Sharing over IP Network. In *USENIX Annual Technical Conference, FREENIX Track* (2005), Anaheim, CA.
- [6] T. RICHARDSON. *The RFB Protocol*, 2004. Version 3.8.
- [7] W. M. FELTER, T. W. KELLER, M. D. KISTLER, C. LEFURGY, K. RAJAMANI, R. RAJAMONY, F. L. RAWSON, B. A. SMITH, AND E. VAN HENSBERGEN. On the Performance and Use of Dense Servers. *IBM Journal of R&D* 47, 5/6 (2003), 671–688.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *login*, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see <http://www.usenix.org/membership/specialdisc.html>

For more information about membership, conferences, or publications, see <http://www.usenix.org>.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at <http://www.sage.org>.

Thanks to USENIX & SAGE Supporting Members

Addison-Wesley Professional/
Prentice Hall Professional
Ajava Systems, Inc.
AMD
Asian Development Bank
Cambridge Computer
Services, Inc.
EAGLE Software, Inc.
Electronic Frontier Foundation
Eli Research
FOTO SEARCH Stock Footage
and Stock Photography

GroundWork Open Source
Solutions
Hewlett-Packard
IBM
Infosys
Intel
Interhack
The Measurement Factory
Microsoft Research
MSB Associates
NetApp

Oracle
OSDL
Raytheon
Ripe NCC
Sendmail, Inc.
Splunk
Sun Microsystems, Inc.
Taos
Tellme Networks
UUNET Technologies, Inc.

